SABRETALK

PROGRAMMER'S REFERENCE GUIDE

Release date: 05/13/94

TABLE OF CONTENTS

INTRODUCTION

SABRETALK is a high level or general source language that is easy to learn and use. The SABRETALK compiler is designed to produce programs that run in the control or executive environment called the Transaction Processing Facility (TPF).

The compiler produces, as output, IBM System 370 Assembler language programs that are re-entrant, that is, programs where a single image of the program in main storage can be servicing several requests.

The most economical approach to applications programming is through the use of a high level language. By using SABRETALK, programmers are able to focus their attention on program logic rather than the individual machine characteristics, thereby improving programmer productivity.

Another advantage derived by the use of SABRETALK is the ease with which programs may be modified. This is possible because of the decreased number of coding statements in a given segment and the high degree of readability of these statements; furthermore, if programmers use descriptive names and comments SABRETALK can be nearly self-documenting in many cases. Finally, the time necessary for program debugging can be reduced considerably because of the clarity of the language and the extensive error flagging routines built into the compiler.

PROGRAMMER'S REFERENCE GUIDE ORGANIZATION

The purpose of this guide is to provide programmers with the rules and conventions for coding application segments and/or programs, for subsequent use in the TPF System. In addition special consideration information has been provided for TPFDF users (see Appendix B).

This guide contains nine chapters:

CHAPTER 1: PROGRAM COMPONENTS:

This chapter introduces the basic components and terminology used in the language.

CHAPTER 2: DATA DEFINITION RULES:

This chapter introduces the construction rules of the basic value statements and Data type definitions.

CHAPTER 3: EXECUTABLE STATEMENT RULES:

This chapter introduces the construction of basic statements and production of simple routines.

CHAPTER 4: EXPANDED DATA DEFINITION RULES:

This chapter provides a more in-depth description of the Data Definition rules and coding conventions.

CHAPTER 5: EXPANDED EXECUTABLE STATEMENT RULES:

This chapter provides an in-depth discussion on executable statements, Built-in functions and Macros supported by SABRETALK and the TPF system.

CHAPTER 6: ALTERNATIVE CODING METHODS:

This chapter discusses efficient coding techniques for optimizing programs.

CHAPTER 7: COMPILER OPTIONS:

This chapter discusses a list of compiler options that may be temporarily over-ridden by the programmer.

CHAPTER 8: THE COMPILER IN AN INTERACTIVE ENVIRONMENT:

This chapter discusses interactive programming facilities through the use of an Interactive Environment Monitor (IEM). It also introduces the Syntax Checker.

CHAPTER 9: COMPILER MESSAGES:

This chapter provides a list of compiler messages.

FORMAT OF THIS GUIDE

From this point on, the manual will present examples that will help to clarify the descriptions given and points made. A uniform system of notation will be used, according to the following rules:

A) Vertical sets of dots enclose an example.

```
:       :
:       :
```

B) Braces (sets of vertical circumflex) enclose alternative items from which selection must be made.

```
_       _
|       |
|       |
-       -
```

C) Brackets (sets of vertical parentheses) enclose options that may be omitted.

```
(       )
(       )
```

D) When more than one element is stacked in braces or brackets the programmer may select the element he desires.

```
_                _
| DEC      |     ( ALIGNED )
| DECIMAL  |     ( PACKED  )
-          -
```

E) `...` A series of three periods indicates a variable number of items that may be included in a list.

F) **F07A** Underlining – used to identify hexadecimal alphabet numerals.

G) Upper-case letters, semicolons, colons, single quotes, parentheses and commas represent information that must appear exactly as shown.

H) Words written in lower case letters denote programmer-defined code according to the rules of the specific element. Although the compiler recognizes only upper case, mixed case is used in this Guide so that capitals may be used to stress keywords in examples of statements.

I) The appearance of one or more items in sequence indicates that the items, or their replacements, should appear in the specified order.

J) The notation *BIN, *BS, etc. is used to mean **BIN** type of data, **BIT** type of data, etc.

Throughout this guide, all references to the Transaction Processing Facility will be in an abbreviated form: TPF.

ADDITIONAL SABRETALK PUBLICATIONS

- SABRETALK Installation and Maintenance Guide
- SABRETALK Programmer's Reference Card

CHAPTER 1: PROGRAM COMPONENTS

PROGRAM COMPONENTS

The SABRETALK programs are coded in free form and can consist of:

- Statements
- Comments
- Assembler Instructions

Statement Classification

A SABRETALK program is constructed from basic elements called statements. Within a program, control normally passes sequentially from one executable statement to the next. As shown in figure 1.1 below, statements may be grouped in categories:

Statement Category	Statement Function
Program Structure:	
PROC	To begin programs and internal procedures and programmer-declared functions
END	To indicate the last statement of DO groups, programs, internal procedures and programmer-declared functions
Data:	
DCL	To specify identifiers and their attributes
INCLUSION	TO incorporate pre-coded statements and/or pre-compiled data record descriptions
%INCLUDE	TO include pre-coded statements from a library
%INCLUDEAF	To include pre-compiled data record descriptions from a library
CONST	To initialize a CONSTANT value
Executable:	
START	To indicate the entrance to executable code when storing of register contents, passed from a calling program, is necessary
Assignment	To transfer data
Control	To affect the order of statement execution
GOTO	To branch
IF	To branch conditionally
DO	To flow conditionally
CALL	To transfer control to internal procedures
RETURN	To return control from internal procedures or from programmer-declared functions
Macros	To communicate with the TPF System or with PL/TPF

Figure 1.1 Categories of Statements

General Statement Structure

Statement labels must abide by the rules for identifiers and they must be followed by a colon.

General Format:

```
(Statement Label and)  (Identifying)  (Statement)   Statement
(colon              )  (Keyword    )  (Body     )   Terminator
(                   )  (           )  (          )

:   labl4:                GOTO          labl2             ;   :
:                         DCL        netpay BINARY(15)    ;   :
:                                                             :
```

Comments

Comments included in the text of the program serve to make it more comprehensible and provide a very effective means of documenting program logic. In general, a comment may be inserted anywhere a blank is permitted, except within a character-string.

Comments may precede or follow a statement or they may be placed within the statement itself. The format for a comment is as follows:

```
/*text of the comment*/
```

The beginning and termination of a comment are denoted by the **/*** and ***/** composites, respectively. Comments cannot be terminated with a slash in column 72; column 71 is the last column in a record. Blanks are permitted within the comment text. Any series of characters may be used in the comment text, with the exception of ***/** itself, since this would signify the termination of the comment. There is no length restriction on a comment.

Keywords

Keywords are reserved words that may only be used as:

- statement identifying keywords(including macro keywords)
- data attribute keywords
- miscellaneous keywords
- built-in function keywords
- register-specification keywords.

Statement Identifying Keyword	Data Attribute Keyword	Misc. Keyword	Built-in Function Keyword	Register Specification Keyword	
CALL	ALIGNED	BY	ABS	#RAC	
CONST	AUTO (AUTOMATIC)	ELSE	ADDR	#RGA	
DCL (DECLARE)	BASED	THEN	ALPHA	#RGB	
DO	BIN (BINARY)	TO	BSTM	#RGC	
END	BIT	WHILE	BSTR	#RGD	
GLOBW	CHAR (CHARACTER)	FILL	CASE	#RGE	
GLOBX	CONSTANT		CSTR	#RGF	
GOTO (GO TO)	DEC (DECIMAL)		INDEX	#RDA	
IF	DEC FLOAT		LSTR	#RDB	
PROC (PROCEDURE)	DEF (DEFINED)		MAX	#RG0	
RETURN	ENTRYBLOCK		MIN	#RG1	
START	FUNCTION		MOD	#RG2	
Macros	LAB (LABEL)		NSTR	#RG3	
%INCLUDE	PACKED		NUMERIC	#RG4	
%INCLUDEAF	PIC (PICTURE)		ROUND	#RG5	
	PTR (POINTER)		SHL	#RG6	
			SHR	#RG7	
			SIGN	#RG14	
			VSTR	#RG15	
				#0	#R0
				#1	#R1
				#2	#R2
				#3	#R3
				#4	#R4
				#5	#R5
				#6	#R6
				#7	#R7
				#14	#R14
				#15	#R15

Macro keywords

Macro keywords are names of macro requests. SABRETALK supports:

A) TPF macros (**ENTRC**, **GETCC**, etc.)
B) SABRETALK Macros (**GLOBX**, **GLOBW**, etc.)
C) User-defined macros (**CSERA**, **INBLK**, etc.)

Assembler Language Instructions

Assembler Language instructions may be coded in columns 1 through 72 of card image records, if the BAL option is in effect. Coding an X in column one identifies an unlabeled Assembler Language instruction. Assembler Language instructions having labels must be coded with a period in column one. Comments must be coded with an asterisk in card column one.

```
Card column 1-----|
                  |
                  V
              :    .addtag   MH    R15,data$(R7)    :
              :    X         A     R15,=F'25'       :
              :    X         ST    R15,data$(R7)    :
              :    *  comment.                      :
```

When Assembler Language instructions are inserted between a '**GOTO'** statement and a SABRETALK routine, a SABRETALK Label must be inserted immediately after the '**GOTO**' statement in order to produce the correct branch instruction around the BAL code.

```
                          -------------
                          GO TO BACK;
                    MOVE:
Card column 1-----|
                  V
              :    .MOVERTN  MH    R15,data$(R7)     :
              :    X         A     R15,=F'25'        :
              :    X         ST    R15,data$(R7)     :
              :      BACK:   IF .........            :
              :                 THEN .....   ;       :
```

This facility should be used with caution and should not be employed to correct errors. Since register usage in the BAL code generated is under the control of the compiler, the programmer must be sure that the registers he uses are properly selected. This requires that the program, to which Assembler Language instructions are to be added, be compiled and that the programmer use the generated output to guard against any conflict in the use of registers. Note that compiler modification as well as program modification may cause a difference in register selection and invalidate the Assembler Language instructions.

If the **NOBAL** compiler option is in effect, coding assembler statements will result in a return code of 12 and error SBT0042E being issued. (See Chapter 7, Compiler Options, for more information)

PROGRAM CONSTRUCTION RULES

A) Statements and comments are coded in columns 2 through 71.

- A semicolon denotes the end of a statement.
- A statement may begin in any column.

B) Card sequence numbers, which may be coded in columns 73 through 80, are ignored by the compiler.

C) When it is necessary to continue a statement or a comment on another line and there is a logical break point, the first part of the statement may begin on one line and continue on the next line, beginning anywhere on that next line. An unlimited number of continuation lines may be used for a statement.

```
Card column 2------|        Card column 71------|
                   |                            |
                   V                            V
              :    a = b + c *( d + e ) +           :
              :    f + g *( h + j ) ;               :
              :                                     :
```

But splitting a name, as in the following example, is not recommended.

```
Card column 2------|              Card column 71------|
                   |                                  |
                   V                                  V
             :                                  a = tot    :
             :     al ;                                    :
             :                                             :
```

D) When it is necessary to continue an assignment statement with a literal on another line, write the literal through column 71, and then continue the literals in column 2 of the next line.

```
Card column 2------|        Card column 71------|
                   |                            |
                   V                            V
              :    A = 'BBBBBBBBBB''BBBBBBBBBBBBB    :
              :    BBBBBBBBBBBBBBBBBBBBBBBBB'        :
              :                                      :
```

E) Syntax checked macros:

1) The maximum number of characters accepted per line is 56.
2) When it is necessary to continue a literal on another line, end statement before column 72 with a quote and a comma, then continue on next line beginning with quote in any column.

```
:                                                                  :
:    WTOPC 'DOT=YES','PREFIX=API2','NUM=61','LET=i',               :
: 'TEXT=''BUG ENCOUNTERED WHILE EXECUTING TEST PROGRAM A',         :
:             'PI2611212''';                                       :
```

F) Non-syntax checked macros: When it is necessary to continue a statement on another line, write the statement through column 71 and continue in column 2 of next line.

```
Card column 2                      Card column 71--|
   |                                               |
   V                                              V
:      WTOPC DOT=YES,PREFIX=API2,NUM=61,LET=I,TEXT=BU   :
:  GENCOUNTE RED WHILE EXECUTING TEST PROGRAM API2611   :
```

Format of Programs

A typical format for a program might be:

A) Program name (in a **PROC** statement)

B) **DECLARE** statements
   1) Names of areas
      a) to be referenced, or
      b) containing parts to be referenced
   2) Names and specifications of areas that will become part of the program

C) **START** statement

D) Processing statements

E) **END** (of program) statement

A simple example of a SABRETALK program:

```
:   abc1d0:  PROC;                    :
:            DCL factor BIN;          :
:            START (factor = #R1);    :
:            factor = factor + 1;     :
:            BACKC;                   :
:            END abc1d0;              :
:                                     :
```

The layout of a program need only follow a few rules; however, standardized formatting of statements enhances readability, helping to clarify the logic and data structure of a program. The previous program example above could be written without a standardized format, as follows:

```
:   abc1d0: PROC; DCL factor BIN;   :
:   START (factor=#R1); factor=     :
:   factor+1; BACKC; END abc1d0;    :
:                                   :
```

Throughout this guide, sample programs and statements have been formatted for readability; it is hoped that the programmer will adopt a coding style along these lines.

Figure 1.2 is the Assembler Language coding produced by compiling the above example. The assembler language listing has been re-formatted to conform to the print limitations of this Guide.

```
LOC    OBJECT CODE   ADDR1 ADDR2 STMT   SOURCE STATEMENT
                                    1           PRINT NOGEN
                                    2           BEGIN NAME=ABC1,
                                                      VERSION=D0
                                  551           ALASC L0
000024 4017 0004    00004         556           STH   R1,FACTOR$(R7)
000028 41F0 0001    00001         557           LA    R15,0001
00002C 4AF7 0004    00004         558           AH    R15,FACTOR$(R7)
000030 40F7 0004    00004         559           STH   R15,FACTOR$(R7)
                                  560           BACKC
000038                            564           LTORG
                          00004   565 FACTOR$   EQU   0004
                          00000   566 $TEMPAUT  EQU   0000
                          00008   567 $TEMPDBL  EQU   0008
                          00010   568 $TEMPFW1  EQU   0016
                          00014   569 $TEMPFW2  EQU   0020
                                  570           FINIS
                                  778           END
```

Figure 1.2 Compiler-generated Assembler Language Coding

CHARACTER SETS

The 59 characters used by the compiler consist of:

- Alphabetic characters (including three from the Universal character set)
- numeric characters
- special characters

Alphabetic Characters:

Character	Name	8-Bit Code	Character	Name	8-Bit Code
A		1100 0001	O		1101 0110
B		1100 0010	P		1101 0111
C		1100 0011	Q		1101 1000
D		1100 0100	R		1101 1001
E		1100 0101	S		1110 0010
F		1100 0110	T		1110 0011
G		1100 0111	U		1110 0100
H		1100 1000	V		1110 0101
I		1100 1001	W		1110 0110
J		1101 0001	X		1110 0111
K		1101 0010	Y		1110 1000
L		1101 0011	Z		1110 1001
M		1101 0100	$	dollar sign	0101 1011
N		1101 0101	@	at sign	0111 1100
			#	number sign	0111 1011

Numeric Characters

Character	8-Bit Code	Character	8-Bit Code
0	1111 0000	5	1111 0101
1	1111 0001	6	1111 0110
2	1111 0010	7	1111 0111
3	1111 0011	8	1111 1000
4	1111 0100	9	1111 1001

Special Characters

Character	Name	8-Bit Code
+	plus sign	0100 1110
-	minus sign	0110 0000
*	asterisk / multiplication code	0101 1100
/	slash / division sign	0110 0001
<	'less than' symbol	0100 1100
>	'greater than' symbol	0110 1110
=	equal-to symbol	0111 1110
&	'and' symbol	0101 0000
|	'or' symbol	0100 1111
^	'not' symbol	0101 1111
(	left parenthesis	0100 1101
)	right parenthesis	0101 1101
'	single quote	0111 1101
,	comma	0110 1011
:	colon	0111 1010
.	period	0100 1011
_	break character	0110 1101
	blank	0100 0000
;	semicolon	0101 1110
%	inclusion symbol	0110 1100

Composites

Composites are sets of two or more characters that are reserved. Blanks are not permitted within composites.

- Composite Operators:

```
^<    not-less-than
^>    not-greater-than
^=    not-equal-to
<=    less-than-or-equal-to
>=    greater-than-or-equal-to
||    concatenation
```

- Pointer Qualification Composite:

```
->    arrow symbol
```

- Comment Composites:

```
/*    start of comment
*/    end of comment
```

CHARACTER SET USAGE

Separators

Name	Graphic	Use
left and right parentheses	( )	sometimes used in an expression for enclosing sequences of operands and operators, for specifying information associated with various keywords and for enclosing subscripts
single quote	'	a pair of single quotes is used to enclose bit and character strings, and is used in PICTURE specifications
comma	,	separates elements of a list
period	.	decimal point
break character	_	used within an identifier or label to enhance readability
colon	:	used to separate the statement label from the statement
semicolon	;	indicates the end of a statement

Use of blanks

Blanks are used as characters in character-strings or, with relative freedom, as separators in statements. Blanks are not permitted within identifiers or composites. Identifiers or keywords must be separated from immediately adjacent identifiers or immediately adjacent keywords by one or more blanks. Individual parts of a statement that are not separated by parentheses or some other separator must be separated by blanks.

The Hexadecimal Numerics

When used in hexadecimal notation, the following are recognized as numeric digits:

0 1 2 3 4 5 6 7 8 9 A B C D E F

Miscellaneous Character Sets

The arrow (**->**) symbol is used as a pointer qualifier.

The equal-to sign (**=**) besides being used as a relational operator in an expression is used as an assignment statement operator.

The number sign (#) is used as a prefix character in System Equates, and also for register specification in the **START** statement and in certain macros.

Certain character sets are used to perform specific functions. The character sets that function as operators can be categorized into four groups:

- arithmetic
- logical
- relational
- concatenation.

The Arithmetic operators are:

```
+    Addition or prefix plus
-    Subtraction or prefix minus
*    Multiplication
/    Division
```

The Logical operators are:

```
&    And
|    Or
^    Not
```

The Relational operators are:

```
<    Less-than
=    Equal-to
>    Greater-than
^<   Not-less-than
^=   Not-equal-to
^>   Not-greater-than
<=   Less-than-or-equal-to
>=   Greater-than-or-equal-to
```

The Concatenation operator is:

```
||   Concatenation
```

DATA CLASSIFICATION

Figure 1.3 illustrates the data classification tree and will serve as a general introduction to data classification.

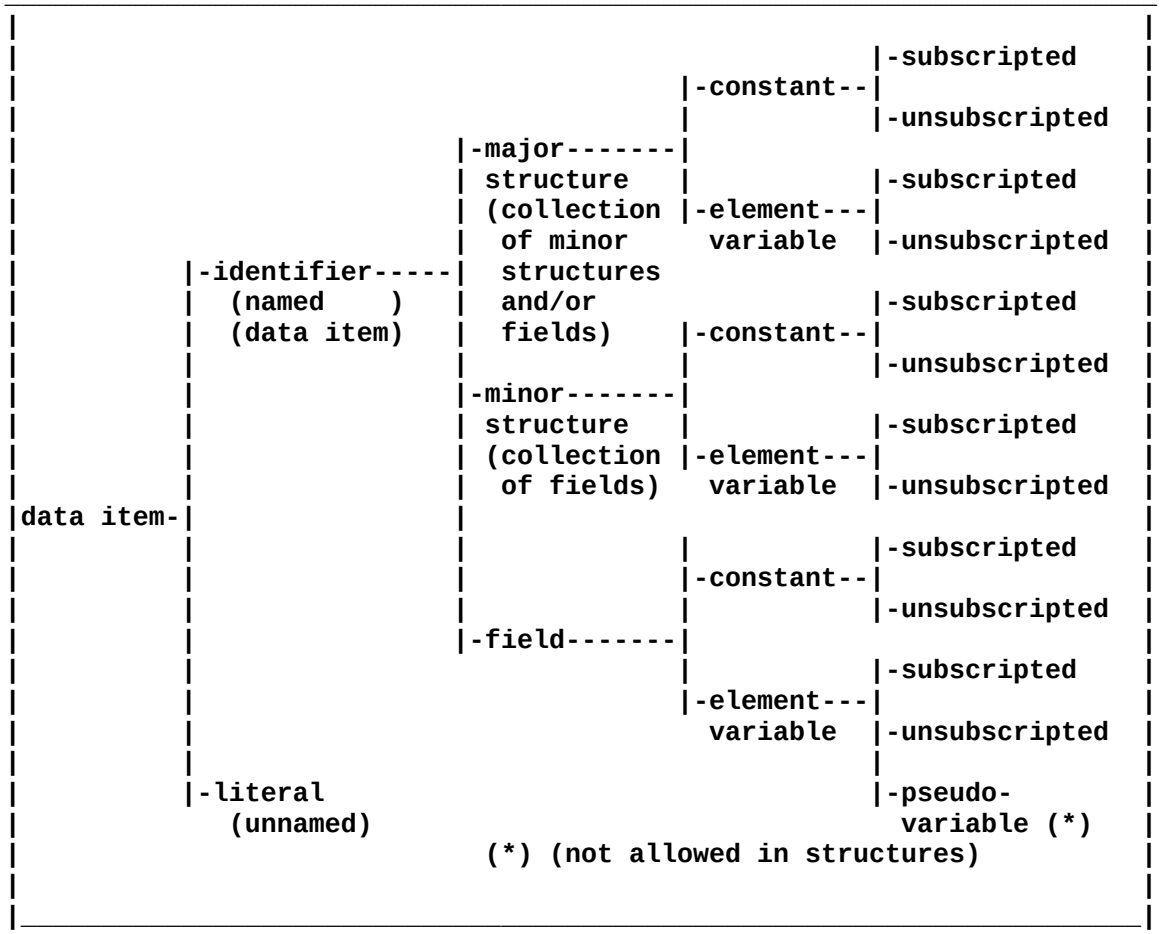


Figure 1.3 Data Classification Tree

Identifiers are names given to data items by the programmer in order that they may be referenced. Identifiers are composed of Alphabetic, Numeric and optional break characters (_). The first character of an identifier must be A through Z, **$**, **@**, or **#**. The break character may appear in the identifier to enhance readability – e.g.:

```
:                 :
:   gross_pay     :
:                 :
```

When coding identifiers, the following rules apply:

A) The first character of an identifier must be alphabetic.
B) All identifiers within a program must be unique.

There is no length restriction on an identifier; however, while the program is being compiled, an identifier greater than eight characters is reconstructed using only the first four and last four characters. For example, the identifier **`flight_ordinal_number`** would be compacted to **`fligmber`** during the compilation process. It is possible for two reconstructed identifiers to appear identical:

```
:    flight_ordinal_number    :
```

```
:                                   :
```

and

```
:     flight_number                 :
:                                   :
```

would both be compacted to become:

```
:     fligmber                      :
:                                   :
```

A) Character # may not be imbedded within an identifier.

The programmer is responsible for ensuring that this compacting does not create duplicate identifiers as illustrated above. The compacted form of the identifier would appear only on the generated assembler output listing, not on the programmer's source code listing.

Whereas identifiers are data names, the names of executable statements are called labels. All of the rules above that apply to identifiers also apply to labels. Labels are used for branching, control being given to the statement whose label is supplied in a **GOTO** statement. It is often necessary to make use of the symbolic address of a labelled executable statement, and so the need for the term 'label' which will be used to mean the name/symbolic address of the executable statement.

```
:          lbl:   b = c;            :
:                                   :
```

DATA CONSTRUCTION METHODS

To explain the construction and the use of statements, a number of definitions, confined necessarily to the context of this guide, are now essential.

The word 'data' is used to mean information, and we shall refer to data or portions of data as data items within the constraints of SABRETALK .

The term '**BIT**', though specifically meant to represent either the binary digit one (1) or the binary digit zero (0), will also be used to mean the core area in which such a digit may be stored.

Certain sets of binary digits are known as numerics, alphabetics, alphanumerics, decimals, binary values, special characters, characters, character-strings; all these terms are self-descriptive.

The usual method of data construction, data reference, data manipulation, etc., is based on the contents of an area. The area size is determined, the data name (identifier) and the characteristics (attributes) of the data that will reside in the area are determined, and then a data statement that explicitly defines the area is prepared as part of the program. If **net_pay** is a **DECIMAL** data item up to five decimal digits, then the programmer may define (i.e., **DECLARE**) the area in which it will reside as follows:

```
:    DECLARE net_pay DEC(5);        :
:                                   :
```

Any data item that is assigned to the area will:

1) conform to the attributes of the area
2) be referenced using the area contents name
3) be assigned an obtainable area address

Variables

The term variable refers to a named area that may be assigned any value that can conform to the attributes.

```
:       DCL   vrble2   BIN ;        :
:                                    :
```

Literals

The facility to define specific values without naming them through the use of identifiers creates a class of values called literals. A literal is a value located within the program and, since it has no name, it is completely defined when it appears in a program statement.

```
:        fld2 = 25;      \           :
:        fld3 = .4 ;      \          :
:        fld4 = '101'B;    \         :
:        fld5 = 'F23C'X;   / Literals:
:        fld6 = 'page';   /          :
:        fld7 = 6.2E23'; /           :
```

A literal, then, is a specific, unnamed, unchangeable value which is defined by a portion of a statement and whose attributes are implicit in the literal. Literal Data is discussed later in Chapter 2.

Constants

The ability to reference, by name, a specific unchangeable value that occupies a specific area is available in SABRETALK. Two statements are needed:

- a data statement that explicitly declares an area into which a named constant is to be placed, defines the attributes of the value and specifies the keyword **CONSTANT**
- a second statement whose keyword is **CONST**, where a specific value and the name of the value are supplied. The value is given in the form of a literal or a **SYSEQ** (System Equate) tag, and a comma is placed after the identifier.

```
:    DECLARE max_pay DECIMAL(5) CONSTANT;    :
:    CONST max_pay, 63210. ;                 :
:                                            :
```

The previous example shows a **DECLARE** statement explicitly defining an area in which can be stored a specific five digit **DECIMAL** value named **max_pay** and it shows the additional statement necessary to initialize the area **max_pay** with a specific value, 63210. decimal. Once initialized, a constant value cannot be changed.

A constant, then, is a named specific value occupying a specified area. The area is declared by one statement and the unchangeable value is initialized by a second. The declaration and initialization of constants is described in detail in Chapter 4.

FIGURE 1.4 is meant to show the statement structure that will be explained in Chapter 2, Data Definition and in Chapter 3, Executable Statements.

```
abc1s1: PROC;
        DCL wt BIN(31);
        DCL msg CHAR(20);
        DCL rwfactor BIN;
        DCL clear PIC '999';
        DCL clearance BIN;
        DCL decr BIN;
        DCL obsthgt BIN;
        DCL i BIN;
        decr = 100;
loop:      DO i = 1 TO 10;
           clearance = wt * rwfactor - obsthgt;
              IF clearance > 0 THEN GOTO fin;
           wt = wt - decr;
           END;
        decr = decr + 500;
        GOTO loop;
fin:    clear = clearance;
        msg = 'clearance = ' || clear || ' feet' ;
        END abc1s1;
```

Figure 1.4 Example of a SABRETALK program

CHAPTER 2: DATA DEFINITION RULES

DECLARE STATEMENT

DECLARE (non-executable) statements are used to define data areas / data fields. Keywords are used to define data attributes (properties associated with the data areas.) There are no restrictions as to the number of **DECLARE** statements and they may be placed before or after any type of statement within the program, except that they must follow the **PROC** statement which names the program. It is good practice to group **DECLARE** statements.

General **DECLARE** Format

```
_          _
| DCL      |  (level)  identifier  ( (dimension) )  data type
| DECLARE  |  (     )              (             )  attribute
-          -                       (             )
(storage   )  (data       ) ... (   ,  (level)  identifier
(class     )  (alignment  )     (     (     )
(attribute)  (attribute   )     (     (     )
                                 (

( (dimension) )  data type  (storage   )  (data       ) ... )   ;
(             )  attribute  (class     )  (alignment  )     )
(             )             (attribute)  (attribute  )     )
                                                          )

                    ( default   storage-class  attribute = AUTO   )
                    ( default   data alignment attribute = PACKED )
```

DECLARE Statement Rules

A) Although labels may be prefixed to **DECLARE** statements, the compiler discards such labels.

B) 'Level' (the numeric degree of data sub-classification) is a decimal integer from 1 through 255. A level of one (1) is assumed if a level is not provided.

C) Attributes (which specify the form and characteristics of data) must immediately follow the identifier to which they apply (in one **DECLARE** statement).

D) Multiple areas may be declared in one **DECLARE** statement, however, each 'identifier-attribute' set must be separated from the succeeding set by a comma:

```
:    DCL  a  attribute(s)  ,  b  attribute(s)  ,    :
:         c  attribute(s)  ;                        :
:                                                   :
```

will give the same result as:

```
:    DCL  a   attribute(s)  ;    :
:    DCL  b   attribute(s)  ;    :
:    DCL  c   attribute(s)  ;    :
:                                :
```

E) All fields must be declared.

F) A field cannot be initialized in the **DECLARE** statement.

Implicit length

DECLARE function to allows implicit length to be coded by referring to a previously declared item's identifier.

Syntax:

```
DECLARE tagname type (size);
DECLARE tagname2 type (tagname);  <---- size of tagname
```

Note: **tagname** must be declared prior to being used as an implicit length.

Valid for **BIN**, **BIT**, **CHAR**, **DEC** and **DEC FLOAT** data types:

```
DCL  BLOCK  CHAR(1055);
DCL  BUFFER  CHAR (BLOCK);  <---- becomes CHAR(1055)
```

Examples:

```
%INCLUDE DT0CA,DTCA28;
DCL 01 DT0CA BASED (DCTAPTR),
    02 DCTAHDR,
       03 ADCTA CHAR(22),
       03 DDCTA CHAR(ADCTA);   <---- becomes CHAR(22)
DCL 01 DC1TA BASED (DCT1PTR),
    02 DCT1HDR,
       03 ADCT1 CHAR(80),
       03 DDCT1 CHAR(ADCT1),   <---- becomes CHAR(80)
       03 DDCT2 CHAR(DDCTA),   <---- becomes CHAR(22)

DCL A_BLOCK CHAR (1055);
DCL BLK_BUFFER CHAR (A_BLOCK) BASED(BLKPTR);  <---  CHAR(1055)

DCL I  BIN(31);
DCL J  BIN(I);    <---- becomes BIN(31)

DCL BTST BIT(16);
DCL BBDEF(4) BIT(BTST);    <---- becomes BIT(16)
```

```
DCL A_LONG_NAME CHAR(80);
DCL NAMETEST CHAR(A_LONG_NAME) DEFINED BLK_BUFFER; <--- CHAR(80)

DCL DEC7V3 DEC(7,3);
DCL DEC9V5 DEC(9,5);
DCL DECTEST DEC(DEC9V5,DEC7V3);    <---- becomes DEC( 9, 3)
DCL DECTEST DEC(11 , DEC9V5);    <---- becomes DEC( 11, 5)
DCL DECTEST DEC(DEC7V3, 4);    <---- becomes DEC( 7, 4)


DCL CALCA DEC FLOAT (16);
DCL CALCB  DEC FLOAT (CALCA);   <---- becomes DEC FLOAT(16)
DCL CALC1 DEC FLOAT (6);
DCL CALC2 DEC FLOAT (CALC1);    <---- becomes DEC FLOAT(6)

DCL INCTEST CHAR(DDCT1);        <---- becomes CHAR (22)
```

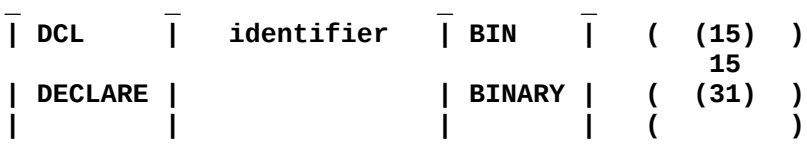
DATA TYPES

The discussion of data types attempts to point out why one might select a given data type over another for a particular data item. The programmer should also reference Chapter 3, Expressions and Data Conversions, and Chapter 3 Assignment statements, as an aid for determining whether the most efficient data type for a given situation has been selected by the programmer.

BINARY Data

Data items are declared **BINARY** by coding the data attribute **BIN** or **BINARY** in the **DECLARE** statement. **BINARY** data is either 16 or 32 bits in length. The high order bit is the sign indicator. The number of numeric **BINARY** digits may be specified in the **DECLARE** statement as either (15) or (31) by placing the desired number in the parentheses following the keyword. If not specified, the default value is 15. The number of digits may also be specified by referring to a previously declared identifier (see Implicit length above).

General declare format for **BINARY** data:




```
:    DCL seat_count BIN (15);          :
:    DECLARE rev_miles BIN (31);       :
:    DECLARE miles BIN (rev_miles);    :
:                                      :
```

In the example:

seat_count	is sixteen bits in length
rev_miles	is thirty-two bits in length
miles	is the same (implicit) length as the data item **rev_miles**, thirty-two bits in length.

Each data item has a sign indicator as the high order bit and is stored in fixed-point **BINARY** format. If **BINARY** data must reside in a field that is not 16 or 32 bits in length, the field may not be declared as **BINARY**. The field should be declared as **BIT** (bit-string) data of the required size (see below).

An example of the assignment of a value (in this case a **BINARY** literal) to a **BINARY** field is as follows:

```
:   seat_count = 50;   :
:                      :
```

The internal representation of **seat_count** in hexadecimal is 0032.

BIT-String Data

BIT-string data items are declared by use of the data attribute **BIT**. The number of bits is specified by a decimal integer, enclosed in parentheses, with a value of 1 through 32. The number of bits may be specified by using the identifier of a **BIT** field previously declared. **BIT**-strings are always treated as unsigned.

General **DECLARE** Format for **BIT**-string Data:

```
_          _
| DCL      |   identifier  BIT (n)   ;
| DECLARE  |
|          |
-          -
               (where n = number of bits: (0 < n <= 32) )

:    DECLARE code_chk BIT(8);    :
:    DCL flight_sw BIT(1);       :
:    DCL fileaddr BIT(32);       :
:    DCL chk2 BIT(code_chk);     :
:                                :
```

In the example above:

code_chk	is eight bits (one byte) in length
flight_sw	is one bit in length.
fileaddr	is thirty-two bits (four bytes) in length
chk	is the same implicit length **code_chk**, eight bits (one byte) in length.

None of the data items is signed; however, in circumstances where positive arithmetic values are required, **BIT** may be used. For example, if only one byte is available for a counter, the counter may be declared as **BIT (8)**. The value is treated as unsigned.

An example of the assignment of a value (in this case a **BIT**-string literal) to a **BIT**-string data item is as follows:

```
:   code_chk = '80'x;   :
:                       :
```

The internal representation of **code_chk** in hexadecimal is 80.

Use of **BIT**-strings in arithmetic expressions is as efficient as **BINARY** under any of the following conditions:

A) Length is 8 bits and field is byte aligned.
B) Length is 16 bits and field is half-word aligned.
C) Length is 32 bits and field is full-word aligned.

Considerably more code will be generated when the length of the **BIT**-string is not a multiple of 8.

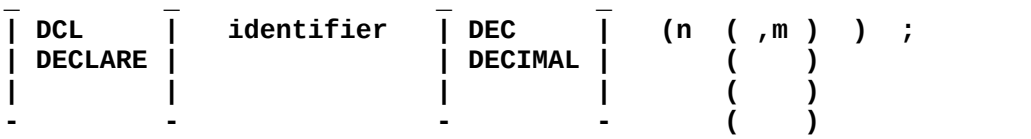
DECIMAL Data

When declaring **DECIMAL** data items, the data attribute **DECIMAL** or **DEC** is used. Following this keyword, the number of digits and the assumed decimal point placement are specified by two decimal integers, enclosed in parentheses, and separated by a comma. The first integer specifies the total number of digits and must be a decimal integer 1 through 15.

The second integer specifies the number of digits to the right of the assumed decimal point and must be a decimal integer from 0 through 15. If the second integer is zero, the integer and preceding comma may be omitted, in which case the decimal point is assumed to be to the right of the rightmost digit. Either digit or both digits may be replaced by identifier names of prior-declared **DECIMAL** items. **DECIMAL** data will be stored in the packed decimal format.

Programmers should develop the habit of declaring **DECIMAL** fields with an odd number of total digits. If an even number of digits is specified, the number will be rounded upward to the next odd number; e.g. a **DEC(4,2)** is treated as a **DEC(5,2)**.

General **DECLARE** Format for **DECIMAL** Data:



(where n = total number of integer and fractional digits (0 < n <= 15))

(where m = number of fractional digits (0 <= m <= 15))

(m <= n)

```
:   DECLARE fare DEC(7,2);            :
:   DCL average DEC(3);               :
:   DCL unround DEC(7,3);             :
:   DECLARE cost DEC(fare,unround);   :
:                                     :
```

In the example:

fare has 7 digits, the last 2 of which are fractional, or to the right of an assumed decimal point.
average has 3 digits, all of which are to the left of an assumed decimal point.
unround has 7 digits, the last 3 of which are fractional, or to the right of an assumed decimal point.
cost has the same implied length as **fare** – 7 digits, but the same assumed number of fractional digits as **unround**, so the last 3 digits are to the right of an assumed decimal point. (See Implicit length section above for additional examples of implicit length for **DEC** type.)

For each of the fields, data would be stored in the packed decimal format.

An example of the assignment of a value (in this case a **DECIMAL** literal) to a **DECIMAL** field is as follows:

```
:   fare = 468.10;   :
:                    :
```

The internal representation of fare in hexadecimal is 0046810C

The **DECIMAL** attribute should not be used without specific reason. It should be used whenever fractional values are involved, or when the field is involved in arithmetic operations and the use of **DEC** for a particular field would provide the most efficient code overall. Certain arithmetic operations, using either mixed data types or **DECIMAL** data items having different assumed decimal points, can cause expensive conversions. Refer to Literal Data, this chapter, for additional information.

DECIMAL FLOAT Data

When declaring Floating Decimal data items, the data attribute **DEC FLOAT** or **DECIMAL FLOAT** is used. Following this keyword, the number of digits of precision specified by one decimal integer, enclosed in parentheses. The precision must be either 6 (short form) or 16 (long form). The identifier of a previously declared **DEC FLOAT** item may also be used. The short form is stored as four bytes of storage, and the long form as eight bytes. Internal representation is always left justified; e.g., **DEC FLOAT (6)** is short form and **DEC FLOAT (16)** is long form. **DECIMAL FLOAT** data is stored as a signed hexadecimal fraction and an unsigned seven bit binary integer called the characteristic.

General **DECLARE** Format for Decimal Floating Data:

```
_          _             _                _
| DCL      |   identifier  | DEC FLOAT      |     (p   )  ;
| DECLARE  |               | DECIMAL FLOAT  |     (    )
|          |               |                |     (    )
-          -               -                -     (    )

               (p = precision   short (6)  or  long (16) )

:   DECLARE fare DEC FLOAT(16);     :
:   DCL average DEC FLOAT(6);       :
:                                   :
```

In the example:

fare has a maximum of sixteen decimal digits of precision.
average has a maximum of six decimal digits of precision.

The data item may represent a decimal number with more than sixteen digits. This is done through the use of an exponent in scientific notation. The coded format must contain a decimal point and an E for exponent followed by a one or two digit exponent. Both the number and the exponent may be signed.

An example of the assignment of a value (in this case a **DECIMAL FLOAT** literal) to a **DECIMAL FLOAT** field is as follows:

```
:   fare = -425.00E-02;     :
:   average = 4.685E45;     :
:                           :
```

The internal representation of **fare** in hexadecimal is C144000000000000

The **DECIMAL FLOAT** attribute should not be used without specific reason. It should be used whenever very large or small fractional values are involved. Certain arithmetic operations using mixed data types can cause expensive conversions.

Refer to Literal Data, this chapter, for additional information.

CHARACTER String Data

CHARACTER string data items are declared by the use of data attribute **CHAR** or **CHARACTER**. Length is specified by a decimal integer, enclosed in parentheses, with a value of 1 through 4087. The length may be implicitly specified by using the identifier of a previously declared **CHAR** item.

General **DECLARE** Format for **CHARACTER** String Data:

```
| DCL     |   identifier   | CHAR      |   (n)   ;
| DECLARE |                | CHARACTER |
                                        (n = number of characters
                                                 (0 < n <= 4087))
```

```
:    DECLARE text CHARACTER(27);     :
:    DCL msg CHAR (4);               :
:    DCL block CHAR (1055);          :
:    DCL buffer CHAR (block);        :
:                                    :
```

In the example:

text	is 27 bytes in length
msg	is 4 bytes in length
block	is 1055 bytes in length
buffer	is the same implied length as **block** - 1055 bytes in length

For each of the fields, data would be stored in the EBCDIC format.

An example of the assignment of a value (in this case a **CHARACTER** string literal) to a **CHARACTER** field is as follows:

```
:   msg = ' OK ';   :
:                   :
```

The internal representation of **msg** in hexadecimal is **40D6D240**.

Refer to Literal Data, this chapter for a discussion of **CHARACTER** string literals.

Numeric Character string Data

Numeric Character string data items are declared by use of the **PICTURE** or **PIC** data attribute together with a **PICTURE** specification.

General **DECLARE** Format for Numeric Character-string:

```
| DCL     |   identifier   | PIC     |  'PICTURE          ;
| DECLARE |                | PICTURE |   specification'
```

Picture Specification for Numeric Character-strings

A **PICTURE** specification is composed of a series of **PICTURE** specification characters enclosed in a pair of single quotes. The **PICTURE** specification characters, for Numeric Character-strings, are **9** and **V**. They are used to specify a digit position and the position of the assumed decimal point, respectively. The **V** serves as a separator for the integer and fractional portions if both are present. If a **V** is not present, the decimal point is assumed to be to the right of the rightmost digit.

A **PICTURE** specification for a Numeric Character-string may consist of from 1 to 15 digits (**9**'s) and, optionally, 1 assumed decimal point (**V**). Repetitive coding of a **PICTURE** specification character can be avoided by preceding it by the number of times, in parentheses, it is to be duplicated. The **PICTURE** specification '**999999V999**' may be coded as '**(6)9V(3)9**' .

The letter **V** does not contribute to the length assigned to the Numeric Character-string. An actual decimal point will not be inserted; rather, alignment of the integer and fractional fields will occur on the position of the **V** in the string when data is stored in the field. Numeric Character-strings can only contain values that are solely numeric in nature. The values are stored internally in the zoned decimal format.

```
:    DECLARE flight PICTURE '9999';    :
:    DCL wages PIC '9999999V99';       :
:                                      :
```

In the example:

flight	is allocated four integer and zero fractional zoned decimal digit positions. The length attribute of **flight** is four.
wages	is allocated seven integer and two fractional zoned decimal digit positions. The length attribute of **wages** is nine.

An example of the assignment of a value (in this case a **DECIMAL** literal) to a Numeric Character-string field is as follows:

```
:   wages = 893.29;   :
:                     :
```

The internal representation of wages in hexadecimal is **F0F0F0F0F8F9F3F2F9**.

There are no Numeric Character string literals.

The major use of Numeric Character-strings is for data that will be used in input/output type operations. They may be coded in arithmetic expressions, however, most arithmetic operations will be considerably less efficient than similar operations performed on data with the **DECIMAL** or **BINARY** attributes. The following example serves to illustrate this point:

```
:    DCL pass_count PIC '999';         :
:    pass_count = 0.;                  :
:        ...                           :
:    pass_count = pass_count + 1. ;    :
:                                      :
```

The internal storage representation of **pass_count** is in the zoned decimal format. In the first assignment statement the packed decimal literal **0.** is unpacked and stored into **pass_count** in the zoned decimal format. In the second assignment statement, **pass_count** is packed and the addition is performed. The result of this addition is unpacked and stored into **pass_count**. If **pass_count** had been declared **DECIMAL (3,0)**, the packing and unpacking instructions would not have been necessary.

If the assignment had not contained a decimal point:

```
:    pass_count = 0;              :
:                                 :
```

then a **BINARY** value would have been specified and a conversion to **DECIMAL** would have occurred, making the conversion even more inefficient.

Please note that provision must be made in the **PICTURE** specification of the receiving field for at least one of the integer or fraction digits of the sending field. Invalid coding will otherwise be produced. For example:

```
:     DCL passcount PIC 'V999';    :
:     passcount = 333;             :
```

```
:                                   :
```

Examples of Numeric Character string assignments:

Source data	PICTURE specification characters	Resultant value
123	999	123
1.2	9V9	12
.02	9V9	00
1.234	V9999	2340
.1234	V9999	1234
12.345	99999	00012

Edited Character-string Data

Edited Character-strings and Numeric Character-strings represent the result of source data being shifted, truncated and padded to fit the pattern of a **PICTURE** specification. An Edited character-string is allowed a variety of additional characters in its **PICTURE** specification so that the **PICTURE** specification, as an editing tool, can be used to insert, append and suppress characters.

Edited character-string variables are declared with the **PICTURE** or **PIC** data attribute. They allow the programmer to exercise greater control over his output data formats. They are coded only as receiving fields in assignment statements and their primary purpose is to produce printable output fields. It must be emphasized that any time an edit operation is performed, one more character than specified (disregarding the **V** character) is used. For example, the **PICTURE** specification '**ZZ99**' will require five bytes of core, rather than four as specified.

General **DECLARE** Format for Edited Character-string Data

```
_          _              _          _
| DCL      |   identifier  | PIC      |  'PICTURE          ;
| DECLARE  |               | PICTURE  |   specification'
-          -               -          -
```

Packed arithmetic data assigned to Edited character-string variables will be converted to the zoned format. Editing cannot be specified for character-string data. Data assigned to Edited character-strings must be arithmetic, containing only digits and, optionally, a sign and an assumed decimal point. (Other characters, even though they may be coded in the **PICTURE** specification, are not allowed in the data to be assigned.) Floating Point data may only be assigned to a floating point Edited character-string.

For both Numeric Character-string and Edited Character-string fields, repetitive coding of a symbol can be avoided by preceding the symbol by the number of times it is to be duplicated. This number must be enclosed in parentheses:

```
:    '999999' can be coded as '(6)9'              :
:    '******V999' can be coded as '(6)*V(3)9'     :
:                                                 :
```

PICTURE Specification for Edited Character-strings

The characters used in **PICTURE** specifications for Edited character-strings may be categorized as follows:

A) Digit specifier / decimal point specifier
B) Suppression characters.
C) Insertion characters.
D) Signs and the currency symbol.

E) Credit and debit characters.
F) Exponent character.

The **PICTURE** characters that are contained in the preceding groups may be coded in many combinations. The **PICTURE** specifications for Edited character-strings may consist of several parts including the sign specification, integer and fractional sub-fields. Each of these sub-fields must contain a minimum of one character which specifies a digit, such as a **9**, **Z**, or *****.

The maximum length of the **PICTURE** specification is 32 characters. Due to the fact that the largest arithmetic field that can be assigned to an Edited character-string consists of 15 digits, a maximum of 15 of the 32 **PICTURE** specification characters may specify digit positions. The remaining characters may only be insertion type characters.

Suppression Characters

To enhance the readability of printed arithmetic values, it is often desirable to replace leading zeros with blanks. The letter **Z** and the asterisk (*****) represent conditional digit positions and will cause leading zeros to be replaced by blanks or asterisks.

The letter **Z** causes leading zeros to be replaced by blanks. When the specified digit position does not contain a leading zero, the digit will not be replaced. **Z** cannot be specified in the same sub-field as *****, nor can it appear to the right of the **9** or any drifting character.

The use of the asterisk (*****) is the same as that of the **Z**, except that an asterisk, rather than a blank, will replace the leading zeros.

Examples of Suppression Characters in Edited Character-strings

Source data	PICTURE specification	Resultant value
001000.	ZZZ999	bbb1000
12321.	ZZ999	b12321
00001.	ZZZZZ	bbbbb1
00000.	ZZZZZ	bbbbbb
19.4	ZV999	b9400
8	ZZ9V99	bbb800
00000	ZZZVZZ	bbbbbb
001.00	**9V99	***100
000.01	***V**	*****1

If a **Z** or ***** appears to the right of the assumed decimal point **V**, then all the digit positions in the **PICTURE** specification must be **Z**'s or *****'s, respectively. Fractional zeros will be suppressed if all fractional digits are zeros and all integer digits have been suppressed. This will result in the entire data item being replaced by blanks or asterisks.

Insertion Characters

The comma (**,**), period (**.**), slash (**/**), and blank (**B**) are insertion characters which cause the specified character to be placed into the associated position of the data. Although insertion characters do not specify digit positions, they are placed between digits and therefore occupy a character position in the Edited character-string. The comma, period, slash, and blank may be suppressed if they are part of a string of suppression characters. Suppression characters are not used to specify arithmetic characteristics.

The comma (**,**) specifies that a comma will appear in the position specified when no zero suppression occurs. When zero suppression does occur, the comma is inserted only when a significant digit is to the left

of it or when the **V** character is coded immediately to the left of it and a fractional value appears. If zero suppression does occur, the comma will be replaced according to the following rules:

A) If the first character to the left of the comma is an asterisk, an asterisk replaces the comma.
B) If this same character is a drifting sign or dollar sign, the comma is considered part of the drifting string.
C) If the first character to the left of the comma is a **Z**, the comma is replaced by a blank.

The period (**.**) specifies that a period will appear under the same conditions governing the appearance of the comma. The period will not cause decimal point alignment of variables assigned to the Edited character-string. This can be accomplished only by use of the **V**. Decimal alignment will occur on the **V**, even if the period appears elsewhere in the **PICTURE** specification.

The slash (**/**) specifies that a slash will appear with all of the governing conditions described above under insertion characters.

The blank (**B**) specifies that a blank will be inserted in the corresponding position with all governing conditions described above under insertion characters.

Unlike the character **V**, which may be coded only once in the **PICTURE** specification, the comma, period, slash, and blank may appear more than once. This allows digits to be separated within the data item assigned to the Edited character-string.

Drifting Characters

The Drifting Characters include the dollar sign (**$**), and the sign characters. These are the letter s (**S**), the plus sign (**+**) and the minus sign (**-**). The dollar sign (**$**) is used as the currency character. The **S**, the plus sign and the minus sign are used to specify arithmetic signs in Edited character-strings.

It is possible to code these characters in a drifting or a static manner; multiple use of the character indicates that it is to be used in the drifting manner. Characters to be used in the drifting manner must be coded in strings that specify every digit position through which the character may drift. Strings of drifting characters will cause zero suppression similar to that caused by use of the **Z** character; however, the drifting character will always be inserted at the end of the string or just to the left of the most significant digit in the field.

Strings of drifting characters may contain any one of the insertion characters and the **V** character. If the insertion character is coded immediately following the drifting string, it is considered part of the string. If a **V** character is coded immediately following the drifting string, it is not considered part of the string. (As indicated previously, **V** serves as a separator for the integer and fractional sub-fields, if both are present.) If a **V** is coded with drifting characters on both sides of it, the drifting character is the only character, which may appear in the sub-field to the right of the **V**. In this case suppression will occur only if all fractional digits are zero and all integer digits have been suppressed, giving a resultant field containing only blanks. Drifting strings cannot be followed by the suppression characters **Z** or *****, and cannot be preceded by a **PICTURE** specification character which indicates a digit position.

If an insertion character will appear as part of a drifting character string, the following will apply:

A) The insertion character will appear if a significant digit has appeared to the left of it.

B) The drifting character will appear in its place if the position immediately to the right contains the most significant digit in the field.

C) A blank will appear if the most significant digit in the field is more than one character position to the right of the insertion character.

If a string contains the drifting character n number of times, the maximum digits that may appear are n.

If these characters are coded in the static manner, they will always appear in the position specified.

The dollar sign (**$**) is used to specify the appearance of a currency character. If coded in the static manner, the character will appear in the position specified. It must be to the left or right of all digits in the field. When coded in the drifting manner, appearance is governed by the rules specified above for drifting characters.

Examples of insertion characters and dollar sign in Drifting Character-strings.

Source data	PICTURE specification	Resultant value
1234.56	$$$9V.99	$1234.56
25.30	$$$9V.99	~~bb~~$25.30
1234.56	$,$$9V.99	$1,234.56
234.56	$,$$9V.99	~~bb~~$234.56

When using the sign characters (**S + -**), the string should always specify an odd number of digit positions (excluding the insertion character), in order to insure the desired result.

Only one of the sign characters may be coded in an Edited character-string. When used in the static manner, the character must be to the left or right of all digits in the field.

- The **S** may be used in the static or drifting manner. It specifies that a plus sign (+) will appear if the number is zero or greater, otherwise a minus sign (-) will appear.
- The plus sign (+) specifies a plus will appear if the number is zero or greater, otherwise a blank will appear.
- The minus sign (-) specifies a minus will appear if the number is less than zero, otherwise a blank will appear.

Examples of the sign characters in Drifting Character-strings.

Source data	PICTURE specification	Resultant value
1234	----9	~~bb~~1234
-1234	----9	~~b~~-1234
1234	++++++9	~~bbb~~+1234
1234.56	++,++9V.99	~~b~~+1,234.56
-1234.56	++,++9V.99	~~bb~~1,234.56

Credit (**CR**) and Debit (**DB**) Composites

The credit and debit composites may only appear to the right of all digits in the Edited character-string. The string should always have an odd number of digit positions when these characters are coded, otherwise they will appear regardless of the value of the field. They may not be coded if any other sign character is present. If an even number of decimal digits is desired, the source field must be adjusted so as to align the first digit of interest on a byte boundary. The 'edit' operation always starts on a byte boundary.

- The credit composite (**CR**) specifies that the field positions will contain the characters CR if the number is negative, otherwise two blanks will appear.
- The debit composite (DB) specifies that the field positions will contain the characters DB if the number is negative, otherwise two blanks will appear.

Floating point Edited character-string.

Floating point data may only be assigned to a special Edited character string. The Edited character string MUST have a **V.** or a **.V** indicating the position of the assumed decimal point and MUST end in **E99** or **ES99**. Example:

```
DECLARE FPECS  PIC'$$,$$9V.99ES99'
```

LABEL Data

LABEL data items are declared by use of the data attribute **LAB** or **LABEL**. They are variables to which are assigned statement labels (the program-relative address of the label is actually assigned) or other label variables.

General **DECLARE** Format for **LABEL** Data:

```
_          _              _        _
| DCL      |   identifier   | LAB    |  ;
| DECLARE  |                | LABEL  |
-          -                -        -

:    DECLARE branch LABEL;    :
:    DCL labvar LAB;          :
:                             :
```

LABEL variables are allocated a half word of **AUTOMATIC** storage.

```
:   DCL labvar LABEL;    :
:          ...           :
:   label2: a = b;       :
:          ...           :
:   label3: b = c;       :
:          ...           :
:   labvar = label2;     :
:   GOTO labvar;         :
:                        :
```

In this example **labvar** is declared a label variable and has the program-relative address of **label2** assigned to it. Execution of the '**GOTO labvar**' statement will now cause control to be transferred to the statement whose label is **label2**. That statement will be executed and control will continue sequentially. Elsewhere in the program **label3** may be assigned to **labvar**, in which case a '**GOTO labvar**' statement would transfer control to the statement whose label is **label3**, etc. The value assigned to **labvar** remains there until another value is assigned to it. Unpredictable results are obtained when a '**GOTO labvar**' statement is executed and **labvar** has never been set.

Label variables may not have **PROCEDURE** statement labels assigned to them.

POINTER Data

POINTER data items are explicitly declared by use of the data attribute **PTR** or **POINTER**. They are variables in which addresses, which refer to (point to) data items, can be stored. Pointers may be declared either explicitly or implicitly.

General **DECLARE** Format for **POINTER** Data (Explicitly Declared):

```
_          _              _          _
| DECLARE |   identifier   | POINTER | ;
| DCL     |                | PTR     |
-         -                -         -

:   DECLARE rptr POINTER ;   :
:   DCL input PTR ;          :
:                            :
```

Implicit declaration involves use of the **BASED** data attribute, which is discussed in depth later in Chapter 4.

General **DECLARE** Format for **POINTER** Data (Implicitly Declared):

```
_          _
| DECLARE |  identifier   attribute     BASED  (identifier)   ;
| DCL     |
_         _

:   DECLARE passdata CHAR(1) BASED (passptr) ;   :
:   DCL output BIN(31) BASED (outptr) ;          :
:                                                :
```

In this example **passptr** and **outptr** are defined as pointers not by the specific use of the **POINTER** attribute but by the parenthesized name following the **BASED** attribute.

Use of either method of declaration will cause one full word of **AUTOMATIC** storage to be allocated for the pointer. If a pointer is to be used to address a data block, the allocation of the data block to be used in conjunction with the pointer is accomplished by use of a macro (e.g., the TPF macro GETCC, etc).

It is the programmer's responsibility to initialize the pointer to contain the address of the data block he wishes to reference after storage allocation has been accomplished. Pointers may be used to reference data blocks and other areas of memory. If the pointer is recognized as being based in the ECB then the compiler will generate the necessary initialization code.

More than one data item may not be BASED on the same pointer:

```
:   DCL a PIC '9' BASED (aptr) ;   :
:   DCL b CHAR(1) BASED (aptr) ;   :
:                                  :
```

In order to bypass this restriction and yet, achieve the same end, do the following:

```
:   DCL a PIC '9' BASED (aptr) ;   :
:   DCL b CHAR(1) DEFINED a ;      :
:                                  :
```

Literal Data

Literals are specific unnamed data values that are defined where they appear in a statement. Literals sometimes appear as parts of executable statements. Literals may be of the **DECIMAL**, **BINARY**, **BIT**-string or **CHARACTER**-string data type. We present them now just before the consideration of executable statements.

BINARY literals are coded as a maximum of 15 decimal numbers without a decimal point; an optional sign may be coded: the maximum value being $2^{31} - 1$, the minimum value being -2^{31}. Internally, all **BINARY** literals are established by the compiler as a positive fullword (**BIN(31)**) number. If the programmer codes a negative **BINARY** literal, the minus is interpreted as a prefix operator and code is generated for the prefix minus operation.

Note: The method of spacing digits is for the sake of clarity, there are no internal blank spaces in values.

BINARY literal	Internal storage bit representation
12	00000000 00000000 00000000 00001100

BIT-string literals are coded as a maximum of 32 binary or as a maximum of 8 hexadecimal digits, enclosed in a pair of single quotes, and immediately followed by the letter B or X, respectively. No blanks may appear between the closing quote and the B or X.

BIT string literal	Internal storage bit representation
'011010'B	00000000 00000000 00000000 00011010
'A3C4'X	00000000 00000000 10100011 11000100

DECIMAL literals are coded as a maximum of 15 signed decimal numbers with a decimal point. A sign may be specified; if not, plus is assumed. If the programmer codes a negative **DECIMAL** literal, the minus is interpreted as a prefix operator and code is generated for the prefix minus operation. The internal representation of a **DECIMAL** literal is packed decimal.

DECIMAL literal	Internal storage Hexadecimal representation
28.	02 8f
28.2	28 2f

DECIMAL FLOAT literals are coded as a mantissa followed by an exponent. The mantissa is a decimal fixed point literal. The exponent is the letter **E** followed by an optionally signed integer, which specifies a power of ten. Decimal Floating-point literals have a precision of sixteen (eight bytes of storage).

DECIMAL FLOAT literal	Scientific Notation
28.E5	$2.8 \times 10^{6} = 2{,}800{,}000.$
2.8E-03	$2.8 \times 10^{-3} = 0.0028$

CHARACTER-string literals are coded as a string containing a maximum of 256 EBCDIC characters enclosed in a pair of single quotes. If blanks are coded as part of the character-string they will become an integral part of the string. A maximum of 4,028 bytes of literals is allowed within a segment. If single quotes are to appear within the string, each single quote must be represented by a composite of two single quotes.

In the following example note that although ‘miami’ and ‘sam's’ are shown as lower case alphabetics, they represent the upper case alphabetics (‘MIAMI’, ‘SAM'S’).

Character-string literal	Internal storage hexadecimal representation
'miami'	D4 C9 C1 D4 C9
'SAM''S'	E2 C1 D4 7D E2

Since there is no method for coding a Numeric Character-string literal or an Edited character-string literal, **DECIMAL** literals are used.

Literal Specification Summary

Data format	Specification	Examples
A) binary	1 through 15 decimal digits without a decimal point. A sign may be specified	123 -53
B) bit-string		
1) bit coding	1 through 32 binary digits enclosed in single quotes and immediately followed by the letter B	'0100100001'B '1011'B
2) hexadecimal	1 through 8 hexadecimal digits enclosed in single quotes and immediately followed by the letter X	'1234C'X 'F1A3'X
C) decimal	1 through 15 decimal digits with a decimal point. A sign may be specified	123. -52.67
D) decimal floating	1 through 16 decimal digits with a decimal point. A sign may be specified. An exponent E must be coded and a 1 or 2 digit exponent optionally signed	1.E0 -5.23E-12
E) character-string	1 through 256 EBCDIC characters enclosed in single quotes	'msg #2'
F) Numeric Character-string	(none)	
G) Edited Character-string	(none)	

CHAPTER 3: EXECUTABLE STATEMENTS - RULES

EXPRESSIONS AND DATA CONVERSIONS

An expression is a combination of operator(s) and operands(s) used for computing a value. An operator is a symbol designating a process to be performed. An operand is a value residing in a data area or the result of an expression (result/exp). Expression types are determined by the types of operators coded within them. Operator (and hence expression) types are Arithmetic, Logical, Relational and Concatenation.

Expressions are not statements of themselves; they are used within various statements.

```
:   IF count = 0 THEN GOTO error;   :
:   DO WHILE invent < maxinvent;    :
:   area = length * width;          :
:                                   :
```

In the first example, **`count = 0`** is an expression (relational).
In the second example, **`invent | maxinvent`** is an expression (relational).
In the third example, **`length * width`** is an expression (arithmetic).

More generally, operators can be grouped into two basic categories: prefix and infix. A prefix operator is one that precedes an operand that is solely involved in that operation. An infix operator is one that is imbedded between two operands, both of which are involved in that operation. The prefix operators provided are the logical not (**^**), the arithmetic plus (**+**) and the arithmetic minus (**-**). The plus and minus are also used as infix operators.

Prefix operators appear in constructions of the form:

```
_                        _          _           _
| keyword                |   prefix | variable  |
| logical operator       |   operator | literal |
| relational operator    |          | constant  |
| arithmetic operator    |          | result/exp|
| concatenation operator |          |           |
|                        |          -           -
-                        -

:   = - 4   :
:            :
```

Infix operators appear in constructions of the form:

```
_          _          _          _
| variable |   infix  | variable |
| literal  |   operator | literal  |
| constant |          | constant |
| result/exp |        | result/exp |
|          |          |          |
-          -          -          -

:   aaa - bbb   :
:               :
```

For expressions involving several operands and/or operators, refer to 'Priority of Operators', this chapter, which specifies possible requirements for parentheses.

In the following sections that treat expressions, certain symbols are used to indicate the operand data-types (and to indicate the data-type for the result of an expression since it too is an operand.)

Symbol	Operand data-type	Value in field treated as:
STR	structure	structure of one or more levels
BIN	binary	signed binary value
BS	bit-string	unsigned binary value
DEC	decimal	signed packed decimal value
DEC FLOAT	decimal floating	signed floating decimal value
CS	character-string	string of EBCDIC characters
NCS	Numeric Character-string	signed zoned decimal value
ECS	Edited character-string	zoned decimal value with EBCDIC characters interspersed
PTR	pointer	
LAB	label	

Arithmetic Expressions

An arithmetic expression is composed of one or more of the following operators:

Operator	Meaning	Syntax
+	prefix or infix plus	(operand) + operand ( )
-	prefix or infix minus	(operand) - operand ( )
*	multiplication	operand * operand
/	division	operand / operand

Prefix plus and minus assign the algebraic signs, positive and negative, respectively, to the numeric quantity. Positive is attributed by default.

Infix plus (**+**) and minus (**-**) specify addition and subtraction operations, respectively.

An operand may be a variable, literal or constant of any type (subject to restrictions within arithmetic operations. See figure 3.1, 'Results of Arithmetic Operations' for valid data-type combinations).

Data Conversions

Figure 3.1 shows the resulting data types obtained from combining two operands with an arithmetic operator. For prefix operators, the result obtained is equivalent to combining two operands of the same data type in the infix expression. For example, a prefix operator on a **DECIMAL** operand will not cause a conversion; the operation will be performed on the **DECIMAL** operand. A prefix operation performed on a *NCS operand, however, will result in a conversion of that operand to **DECIMAL** (See Chapter 5, Precision).

data type	STR	CS	NCS	ECS	BS	DEC	FLT	BIN	PTR	LAB
STR	-	-	-	-	-	-	-	-	-	-
CS	-	-	-	-	-	-	-	-	-	-
NCS	-	-	DEC	-	DEC	DEC	-	DEC	-	-
ECS	-	-	-	-	-	-	-	-	-	-
BS	-	-	DEC	-	BIN	DEC	FLT	BIN	-	-
DEC	-	-	DEC	-	DEC	DEC	FLT	DEC	-	-
FLT	-	-	-	-	FLT	FLT	FLT	FLT	-	-
BIN	-	-	DEC	-	BIN	DEC	FLT	BIN	BIN	-
PTR	-	-	-	-	-	-	-	BIN	-	-
LAB	-	-	-	-	-	-	-	-	-	-

Key:

Dash (-) indicates illegal combination of data types.

FLT = DEC FLOAT

NOTE: Conversions to or from floating point must must not be based or defined storage.

Figure 3.1: Results of Arithmetic Operations: **+,-,*,/**

Examples of arithmetic expressions and their handling:

```
:   DCL score BIN;            :
:   DCL weight BIN;           :
:   grade = score * weight;   :
:                             :
```

Since both **score** and **weight** are **BINARY**, the result of the multiplication will be **BINARY**.

```
:   DCL totcost  DEC(7,2);           :
:   DCL unitcost DEC(7,2);           :
:   DCL numunits BIN;                :
:   totcost = unitcost * numunits;   :
:                                    :
```

numunits will be converted to **DECIMAL**, then multiplied by **unitcost**, giving a **DECIMAL** result.

```
:   DCL totcost   DEC FLOAT(6);      :
:   DCL unitcost  DEC FLOAT(6);      :
:   DCL unitprice  DEC(7,2);         :
:   DCL numunits BIN;                :
:   totcost = unitcost * unitprice;  :
:   totcost = unitcost * numunits;   :
:                                    :
```

unitprice will be converted to **FLOAT** data, then multiplied by **unitcost**, giving a **FLOAT** result.
numunits will be converted to **FLOAT** data, then multiplied by **unitcost**, giving a **FLOAT** result.

```
:   DCL totcost   DEC(7,2);          :
:   DCL unitcost  DEC FLOAT(6);      :
:   DCL unitprice  DEC(7,2);         :
:   DCL numunits BIN;                :
:   totcost = unitcost * unitprice;  :
:   totcost = unitcost * numunits;   :
:                                    :
```

unitprice will be converted to **FLOAT** data, then multiplied by **unitcost**, giving a **FLOAT** result that will be converted to **DECIMAL**.
numunits will be converted to **FLOAT** data, then multiplied by **unitcost**, giving a **FLOAT** result that will be converted to **DECIMAL**.

Relational Expressions

The following list contains all the relational (comparison) operators that may be used. Several of the operators are composites.

Operator	Meaning	Syntax
<	less-than	operand < operand
>	greater-than	operand > operand
=	equal-to	operand = operand
^<	not-less-than	operand ^< operand
^>	not-greater-than	operand ^> operand
^=	not-equal-to	operand ^= operand
<=	less-than-or-equal-to	operand <= operand
>=	greater-than-or-equal-to	operand >= operand

An operand may be a variable, literal or constant of any type except *ECS (and subject to restrictions within arithmetic operations). See figure 3.2: 'Results of Relational Operations', this chapter, for valid data-type combinations. (See Priority of Operators, this chapter, for possible requirement of parentheses).

Relational Operations

There are two types of relational operations: Algebraic and Logical.

- Algebraic relational operations involve the arithmetic comparison of two signed numeric values.

- Logical relational operations involve the left-to-right character-by-character comparison of one sequential character set with another. If the operands are of different lengths the shorter one is extended to the right with blanks.

```
:    DCL b CHAR(3);                :
:    IF b = 'a' THEN GOTO next;    :
:                                  :
```

The above example results in a comparison of the three bytes of variable **b** with the three-byte character set: literal '**a**', blank, blank.

data type	STR	CS	NCS	ECS	BS	DEC	FLT	BIN	PTR	LAB
STR	-	-	-	-	-	-	-	-	-	-
CS	-	2	-	-	-	-	-	-	-	-
NCS	-	-	3	-	3	3	-	3	-	-
ECS	-	-	-	-	-	-	-	-	-	-
BS	-	-	3	-	1	3	7	4	-	-
DEC	-	-	3	-	3	6	7	3	-	-
FLT	-	-	-	-	7	7	7	7	-	-
BIN	-	-	3	-	4	3	7	5	-	-
PTR	-	-	-	-	-	-	-	-	1	-
LAB	-	-	-	-	-	-	-	-	-	1

Key:

Dash (-) indicates illegal operation.

FLT = DEC FLOAT

1: Align on right. Zero fill shorter operand on left. Arithmetic compare.

2: Align on left. Blank fill shorter operand on right. Logical compare.

3: Convert operand(s) to decimal. Align on assumed decimal point. Arithmetic compare.

4: Align on right. Zero fill bit operand on left. Arithmetic compare.

5: Align on right. Arithmetic compare.

6: Align on assumed decimal point. Arithmetic compare.

7: Convert operand(s) to floating point. Floating compare.

Figure 3.2 Results of Relational Operations **<, >, =, ^<, ^>, ^=, >=, <=**

The result of a relational operation is a 31-bit **BINARY** number equal to a positive one (1) if the comparison is true, or equal to a positive zero (0) if the comparison is false. The result of the expression may itself be used as an operand, for example, the result of the following expression is two (2):

```
:   ('abc' ^= 'xyz') + ('aaa' = 'aaa')   :
:                                        :
```

Such coding can be confusing, and should be avoided.

Logical Expressions

A logical expression is composed of one or more of the following operators:

Operator	Meaning	Syntax
^	not	^ operand
&	and	operand & operand
|	or	operand | operand

An operand may be a variable, literal or constant of any type subject to the restrictions for logical operations. See figure 3.3 'Results of Logical Operations', for valid data-type combinations.

Use of the 'not' operation specifies that the bits in the operand are to be reversed: the result can be considered as a bit product with no arithmetic carry. The 'or' operation is an 'inclusive or': the result can be considered as a bit overlay. (See Priority of Operators, this chapter, for possible requirement of parentheses).

data type	STR	CS	NCS	ECS	BS	DEC	FLT	BIN	PTR	LAB
STR	-	-	-	-	-	-	-	-	-	-
CS	-	-	-	-	-	-	-	-	-	-
NCS	-	-	-	-	-	-	-	-	-	-
ECS	-	-	-	-	-	-	-	-	-	-
BS	-	-	-	-	1	-	-	1	-	-
DEC	-	-	-	-	-	-	-	-	-	-
FLT	-	-	-	-	-	-	-	-	-	-
BIN	-	-	-	-	1	-	-	1	-	-
PTR	-	-	-	-	-	-	-	-	-	-
LAB	-	-	-	-	-	-	-	-	-	-

Key:

Dash (-) indicates illegal combination of data types.

FLT = DEC FLOAT

1: Align on right. Zero fill on left to 32 bits. Perform logical operation.

Figure 3.3 Results of Logical Operations: **&, |**

Padding

When two operands are combined in a logical expression, both operands are converted or extended to 32 bits in length prior to the (infix) 'and' operation or (infix) 'or' operation. In the case of the (prefix) 'not' operation, the operation is performed prior to extending the field to 32 bits. When fields of unequal lengths are combined in a logical operation they are aligned on the rightmost bit and then extended on the left, with zeros, to 32 bits.

Boolean Arithmetic of Logical Operations

Initial values		Operation result			
A	B	^A	^B	A&B	A|B
0	0	1	1	0	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	1	1

```
:   ^ '010111'B             yields  '101000'B   :
:                                               :
:   '111111'B & '101'B      yields  '000101'B   :
:                                               :
:   '010111'B | ^ '101'B    yields  '010111'B   :
:                                               :
:   ^ '101'B | ^ '111111'B  yields  '000010'B   :
:                                               :
```

Concatenation Expressions

There is only one concatenation operation provided:

Operator	Meaning	Syntax
||	concatenation	operand || operand

The concatenation operator is a composite of two logical 'or' characters.

An operand may be a variable, literal or constant of any type, subject to the restrictions for concatenation operations. See figure 3.4 'Results of Concatenation Operations', this chapter, for valid data-type combinations. (Refer to, Priority of Operators, this chapter, for possible requirement of parentheses.)

Concatenation specifies that the values of two operands are to be sequentially ordered; that is, the last character or bit of the first string is followed immediately by the first character or bit of the second string. Subscripted *NCS data can not be concatenated.

	OPERAND B									
OPERAND A: data type	STR	CS	NCS	ECS	BS	DEC	FLT	BIN	PTR	LAB
STR	-	-	-	-	-	-	-	-	-	-
CS	-	1	1	1	-	-	-	-	-	-
NCS	-	1	-	-	-	-	-	-	-	-
ECS	-	1	-	-	-	-	-	-	-	-
BS	-	-	-	-	2	-	-	2	-	-
DEC	-	-	-	-	-	-	-	-	-	-
FLT	-	-	-	-	-	-	-	-	-	-
BIN	-	-	-	-	2	-	-	2	-	-
PTR	-	-	-	-	-	-	-	-	-	-
LAB	-	-	-	-	-	-	-	-	-	-

Key:

Dash (-) indicates illegal combination of data types.

FLT = DEC FLOAT

1: Join operand B to operand A. Result is a character-string whose length is equal to the sum of the lengths.

2: Join operand B to operand A. Result is a bit string whose length is equal to the sum of the lengths. Conversion is not performed on binary fields therefore the result may contain two sign bits.

Figure 3.4 Results of Concatenation Operations

The results of the concatenation operation will always be the creation of another string. The resulting string type (**BIT** or **CHARACTER**) depends on the data characteristics of the strings being concatenated.

Bit-string Concatenation:

```
:   '01001'B  ||  '0110'B   yields   '010010110'B   :
:                                                   :
```

Character-string Concatenation:

```
:   'chara' || 'cter string'  yields  'character string'   :
:                                                         :
```

Operands in concatenation operations may not be subscripted.

PRIORITY OF OPERATORS

The evaluation of an expression specifying several operations is performed by establishing operator priorities, as depicted in the following table:

Priority level	Operator(s)
1	prefix +, prefix -, ^
2	*, /
3	infix +, infix -
4	||
5	<, ^<, <=, =, ^=, >=, >, ^>
6	&
7	|

General Operator Priority Rules:

A) Highest priority level (level 1) operations are performed first.

B) If two or more prefix operators (includes the not (**^**) symbol) appear on an operand, they are applied on a right to left basis.

C) If two or more infix operators of the same priority level appear in the expression, they are applied on a left to right basis.

D) The priority of operators within an expression can be altered by the use of parentheses; any operations enclosed in parentheses are performed first. Within parentheses, rules A) through C) apply. When two or more sets of parentheses appear in an expression, treatment is as follows: nested sets are evaluated beginning with the innermost and working outward; non-nested sets are evaluated on a left to right basis.

E) Relational operations may be enclosed in parentheses when used in an **IF** statement (See Chapter 3: **IF** Statements).

Example:

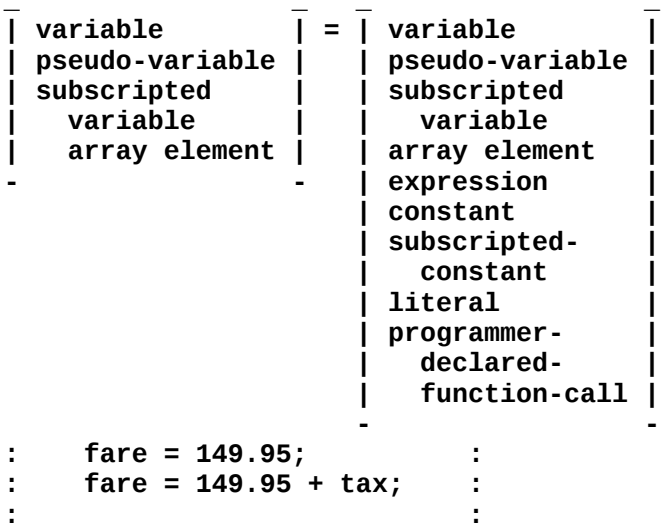
```
:   total = price * (quant + spares) / share;   :
:                                               :
```

The expression **quant + spares** will be evaluated first, then the result will be multiplied by **price**. The result of that multiplication will then be divided by **share**.

ASSIGNMENT STATEMENTS

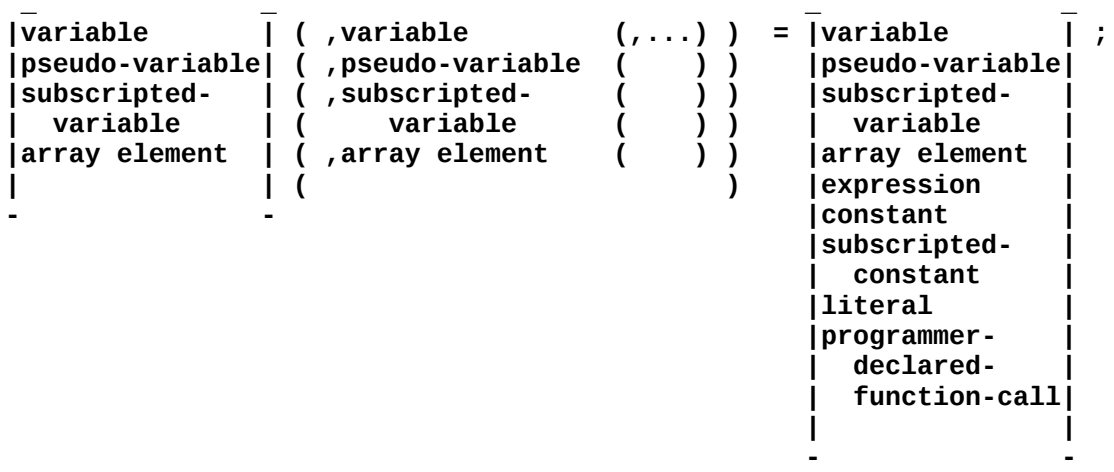
Unlike other types of statements, the Assignment statement does not have a keyword. Instead, the equal sign (**=**), as an assignment operator, denotes assignment. The Assignment statement serves to store (assign) the value of a variable, literal, constant or expression into the storage location of the specified variable.

Simple Assignment




```
:    fare = 149.95;          :
:    fare = 149.95 + tax;    :
:                            :
```

Multiple Assignment




```
:    a, b, c = 3;    :
:                    :
```

This is equivalent to:

```
:       c = 3;       :
:       b = 3;       :
:       a = 3;       :
:                    :
```

Data Conversion, Truncation and Padding

There are two basic types of assignment statements: Arithmetic and Character.

Arithmetic assignment implies a conversion, if necessary, to the data type of the receiving field and then an alignment on the decimal point of the receiving field. Arithmetic operations, using either mixed data types or **DECIMAL** data having different assumed decimal points, cause expensive conversions. The following example illustrates this point:

```
:    DCL  a  BIN;           :
:    DCL  b  BIN;           :
:    DCL  c  DEC(5);        :
:    a  =  c  *  b  +  5;   :
:                           :
```

In the example above, **b** will be converted to **DECIMAL** and multiplied by **c**. The **BINARY** literal **5** will then be converted to **DECIMAL** and added to the result of the preceding multiplication. This final result will then be converted to **BINARY** and stored in **a**.

Since **c** had no fractions, perhaps it could have been declared as **BIN (31)** instead of **DEC (5)**. This would have made the expression much more efficient.

The correct sign bit is always stored. If, as a result of the alignment of the decimal point, the integer portion is too large, the most significant bits are stripped away. If the fractional portion is too large, the least significant bits are discarded. If the receiving field is larger, there is an extension of the sending field with digits that will not alter its algebraic value. In the chart below, the term 'arithmetic move' means assigning a value after any necessary truncation, extension or alignment to a decimal point. The same arithmetic value is always retained.

The programmer should exercise care when moving **BIT**-string data to **BINARY** fields to ensure that the data does not overflow the number portion of the field and set the sign bit. For example, a **BIT(16)** field being moved to a **BIN(15)** field: the high order bit of the **BINARY** field could be set to 1, thereby making the value a negative number. Unless the input data is controlled, it is safer to use **BIN(31)** than **BIN(15)** in such situations.

Character assignment results in alignment of the left end of the sending and receiving fields and a byte by byte assignment. If the sending field is longer, it is truncated; if it is shorter, it is extended with blanks.

Structure Assignment

General Format:

```
structure   (,structure   (,...) ) = structure  ;
            (             (    ) )
            (                   )
```

The only valid use of structure in assignment statements is structure-to-structure assignment. The "receiving" and/or "sending" structure may be a minor or a major structure. Structures cannot be compared, concatenated, added, etc. A structure may only appear in an assignment statement by itself. The address of a structure may be obtained, or a structure may be scanned, through the use of the built-in functions **ALPHA**, **NUMERIC**, **ADDR**, and **INDEX**. There is no such thing as a literal structure. A structure assignment is not performed on an element-by-element basis, but as a single move over the length of the whole structure without conversion. If the length of the structure on the left of the assignment statement is less than the length of the structure on the right, movement of data stops when the smaller is full. Conversely, when the leftmost structure is larger, the remainder of the leftmost structure is blank-filled. The result of the following assignment would be to move 22 bytes from input into output.

```
:   DCL 1 output BASED (outptr),   :
:         2 num BIN(15),           :
:         2 task CHAR(20);         :
:                                  :
:   DCL 1 input BASED (inptr),     :
:         2 calc BIN(15),          :
```

```
:         2 inname CHAR(25);       :
:                                  :
:   output = input;                :
:                                  :
```

General Rule For Structure Assignments: In structure assignments, each variable in the receiving field must represent a structure of arithmetic or string data types.

Figure 3.5 is a table summary of data type usage in assignment statements. Data types for receiving fields are listed vertically; types for sending fields, horizontally. Actions for each combination are performed in the sequence indicated.

	RIGHT OF EQUAL SIGN									
data type	STR	CS	NCS	ECS	BS	DEC	FLT	BIN	PTR	LAB
STR	1,2	-	-	-	-	-	-	-	-	-
CS	-	1,2	-	-	-	-	-	-	-	-
NCS	-	-	13,3	-	4,5,13, 6,3	13,6,3	-	5,13, 6,3	-	-
ECS	-	-	10,13, 7	-	4,5, 13,7	13,7	-	5,13, 7	-	-
BS	-	-	10,8,9, 11,3	-	4,12,3	8,9,11, 12,3	-	8,12, 3	-	-
DEC	-	-	10,13,3	-	4,5,13 3	13,3	@	5,13, 3	-	-
FLT	-	-	-	-	@	@	14,1	@	-	-
BIN	-	-	10,9, 11,3	-	4,3	9,11,3	@	3	1	-
PTR	-	-	-	-	-	-	-	1	1	-
LAB	-	-	-	-	-	-	-	-	-	1

(Left-hand data type column: LEFT OF EQUAL SIGN)

Key: Dash (-) indicates illegal assignment.

At sign (@) indicates Literal or Constant ONLY.

FLT = DEC FLOAT

1: Character move.
2: Blank fill receiving field on right or truncate sending field on right, if necessary.
3: Arithmetic move.
4: Extend on left with zeros to full word, if necessary.
5: Convert to decimal.
6: Unpack.
7: Edit in accordance with PICTURE specification.
8: Take absolute value.
9: Truncate to integer.
10: Pack.
11: Convert to binary.
12: Align on right.
13: Align assumed decimal points.
14: Clear receiving field to hex zero if necessary.

FIGURE 3.5 RESULTS OF ASSIGNMENTS.

Structures can only be assigned to structures. Assignment is by character move, blank fill on right.

When an assignment statement contains non-subscripted references to an array, only the first element is involved.

```
:    DCL  ar(3) CHAR(2) ;    :
:    DCL arr(4) CHAR(3) ;    :
:    arr = ar ;              :
```

The above example is the equivalent of:

```
:    arr(1) = ar(1) ;        :
:                            :
```

In multiple assignments involving subscripted variables, subscripts are frozen prior to the assignment. This means that in the sequence

```
:   i = 2;               :
:   array(i) = i = 3;    :
:                        :
```

the value 3 would be assigned to **array(2)** and not to **array(3)**. The final value of **i**, however, will be 3.

Character-string to character-string assignment

When a character-string of a shorter length is assigned to another character-string of a greater length, the character move and blank fill operation is performed. This operation assigns the smaller string to the larger string (character by character, and proceeding from left to right) until the smaller string has been assigned. The remaining unused bytes are filled with blanks.

```
:    DCL emblem CHAR (12);    :
:    Emblem = 'EAGLES';       :
:                             :
```

The character-string **emblem** would appear in storage as a twelve-character field; six characters of assigned data suffixed with six blanks:

```
(EAGLES      ).
```

Arithmetic to arithmetic (*DEC, *NCS, *ECS, *BIN, *DEC FLOAT)

Assignment of arithmetic types involves an arithmetic move. If data types differ, the data on the right of the assignment symbol is converted to the attributes declared for the variable on the left. Alignment is on the decimal point. The exception to this rule is floating data, which uses a character move.

```
:    DCL fare DEC (5,2);    :
:    fare = 5000.206;       :
:                           :
```

In the example, **fare** would appear in storage as **00020** (with an assumed decimal point fixing the value at .20). Note that the high order digit and the low order digit were truncated to agree with the size and decimal point placement that was declared with fare.

If **fare** had been declared as ***NCS**, conversion would have been necessary in addition to the truncation illustrated above.

```
:    DCL fare DEC FLOAT(16);    :
:    fare = 5.000206E3;         :
:                               :
```

In the above example, **fare** would appear in storage as **44138834BC6A7EF9** with no truncation and the characteristic maintaining the decimal point.

```
:    DCL fare DEC FLOAT(16);    :
:    fare = 5000;               :
:                               :
```

In this example, **fare** would appear in storage as **4413880000000000** with no truncation. The literal would first be represented as a full word **00001388** and then converted to floating point. This is a highly efficient method of assigning integers to floating data.

```
:    DCL fare DEC FLOAT(16);    :
:    fare = 5000.206;           :
:                               :
```

In this example, **fare** would appear in storage as **44138834BC6A7EF9** (with no assumed decimal point). Note that unlike **DEC** no truncation would occur. Conversion would be necessary.

```
:    DCL fare DEC (5,2);        :
:    fare = 5.000206E3;         :
:                               :
```

In this example, the literal would first be converted to floating point format as in the above example. Then it would be converted to the **DECIMAL** equivalent of **5000.20600000000**. Finally, **fare** would appear in storage as **00020** (with an assumed decimal point fixing the value at .20). Note that the high order digit and the low order digit were truncated to agree with the size and decimal point placement that was declared with **fare**. Because floating point numbers have a much larger range, they cause drastic truncation and conversions when assigned to **DECIMAL**. This should be avoided whenever possible.

```
:    DCL fare PIC '999V99';     :
:    fare = 5000.206;           :
:                               :
```

The literal **5000.206** would first be converted (via unpack) to Zoned arithmetic data and then aligned on the decimal point specified in the declaration of **fare**. Again, **fare** would appear in storage as **00020** but it would be in Zoned format instead of Packed Decimal.

When a decimal point is not specified in the declaration, it is assumed to be to the right of the rightmost digit.

```
:    DCL c DEC (7);       :
:    DCL b PIC '999';     :
:                         :
```

In the case of **c**, the decimal point is assumed to be to the right of a seven-digit, signed **DECIMAL** number. In the case of **b**, it is assumed to be to the right of a three-digit Zoned Decimal number. Truncation required for assignment, then, would always be of the high order digits.

The following assignment also involves truncation:

```
:    DCL e BIN (15);     :
:    e = 131071;         :
:                        :
```

The binary representation of the literal would be **00000000 00000001 11111111 11111111**, or 17 binary ones. In this case, the two high order binary 'ones' of the value that is extracted from the 32-bit literal would be truncated and the value assigned to **e** would be equal to 32,767 **DECIMAL**, or **01111111 11111111** binary (the maximum value for a positive half-word number).

The programmer is responsible for insuring that assignment statements and data declarations are properly matched to produce correct results. Whenever possible, it is a good idea to always **DECLARE** an area with no fewer significant digits than any data to be assigned to it (See Chapter 5, Precision).

BIT-string to BIT-string

If a **BIT**-string is assigned to another **BIT**-string, alignment is on the right and the move is a bit-for-bit assignment, right to left. Padding or truncation is performed on the left.

```
:    DCL g BIT (7);       :
:    DCL e BIT (6);       :
:    g = '1100'B;         :
:    e = '111111100'B;    :
:                         :
```

In this example, **g** would appear in storage as **0001100** and **e** would appear as **111100**.

Arithmetic to BIT-string

This move results in the absolute value of the arithmetic data to the right of the assignment symbol being converted to a **BIT**-string. For Numeric character-string values or for **DECIMAL** values, the number is truncated to yield integer values only. Truncation or padding takes place on the left.

```
:    DCL f BIT (12);     :
:    f = -15.2;          :
:                        :
```

The literal **-15.2** would yield the absolute value of **15.2**. It would then be truncated to an integer **15**, and **f** would appear in storage as **000000001111**.

An example of the need for and the use of absolute values follows:

```
:   ce1fm5 = pd1fch;    :
:                       :
```

ce1fm5 is declared as **BIT(32)**. If **pd1fch** (assume that it's a forward chain field containing a file address) has been declared as **BIT(32)**, then the assignment will achieve the intended result. If **pd1fch** has been declared **BIN(31)**, however, the result in **ce1fm5** will not necessarily be the intended one, and therefore the programmer should declare all fields which might contain file addresses as **BIT(32)**.

BIT-string to arithmetic

In this assignment the **BIT**-string is expanded to a full word, if necessary, by padding on the left with zeros before performing the arithmetic move.

```
:    DCL h BIN;      :
:    h = '1010'B;    :
:                    :
```

The **BINARY** literal **1010** will be expanded to:

```
00000000 00000000 00000000 00001010
```

or in hexadecimal:

```
'00000009'x
```

The arithmetic move will align the data on the right (the decimal point is assumed to be at the right of the rightmost bit), and **h** will appear in storage as **00000000 00001010**.

Assignment of Labels and Pointers

A straight character move is used for assignments involving pointers or labels. All pointers are four bytes in length; all labels are two bytes in length. Since the moves would always involve areas of the same size, no operation other than an assignment is required. Replacement would be on a byte-by-byte basis, from left to right.

GOTO STATEMENTS

The **GOTO** statement provides unconditional transfer of control to any labelled statement within the program, with the following exceptions: an internal procedure, an external procedure, or a programmer-declared function.

Statement Syntax:

```
    _   _ _     _
    | GOTO  |   | label constant |  ;
    | GO TO |   | label variable |
    |       |   |                |
    -       -   -                -
```

If a label constant is specified, control is transferred to the statement prefixed by the label.

If a label variable is used, control is transferred to the statement prefixed by the label assigned to the label variable. A label variable will retain a value (label address) until another assignment is made into it. Since different assignments may occur throughout the program, a '**GOTO label_variable;**' might not always transfer control to the same statement.

```
:    DCL switch BIN;                                  :
:    DCL labvar LABEL;                                :
:           ...                                       :
:    testswitch: IF switch = 1 THEN labvar = lbl3;    :
:                               ELSE labvar = lbl4;    :
:                GOTO labvar;                         :
```

```
:          ...                                    :
:         lbl3: a = b ;                           :
:          ...                                    :
:         lbl4: c = d ;                           :
:                                                 :
```

Notes:

A **GOTO** statement should not transfer control from outside an iterative **DO** group to a statement within the loop.

Care should be taken to insure that a **GOTO** invocation does not result in a looping condition that would not terminate, the simplest example of which would be:

```
:    badloop: GOTO badloop;    :
:                              :
```

DO STATEMENTS

The **DO** statement heads a series of statements called a **DO** group. A **DO** group consists of a **DO** statement followed by one or more statements then a terminating **END** statement. The format of the **DO** statement indicates whether the **DO** group is non-iterative or iterative. **NOTE: The maximum number of any combination of DO statements in one program is 193.**

Format of Non-iterative DO Statement

```
DO ;
```

Format of Iterative DO Statement

```
DO   variable   =   specification   ;
     element
```

Where specification is:

```
_          _
|arithmetic | (TO arithmetic  (BY arithmetic ) ) (WHILE arithmetic )  ;
| expression| (    expression (    expression) ) (       expression)
|logical    | (               (              ) ) (      logical    )
| expression| (                                 ) (       expression)
|relational |                                     (      relational )
| expression|                                     (       expression)
|defined    |                                     (      defined    )
| function  |                                     (       function  )
-           -
```

Non-iterative DO Group

Invocation of a non-iterative DO results in a single execution of the **DO** group. Non iterative **DO** groups are typically used in **IF** statements.

```
:    IF   ...   THEN     :
:    DO;                 :
:    a = (b + c) / d;    :
:    f = b - c;          :
:    END;                :
```

```
:                          :
```

Iterative DO Group using WHILE Clause

In this type of iterative **DO**, the repetitive execution of the **DO** group (or **DO** loop) is controlled by the evaluation of the operand in the **WHILE** clause. The execution will be repeated as long as the clause is true, i.e., the **END** statement effects a branch to the top of the loop whenever the **WHILE** clause is true. Once the **WHILE** clause is false, control is passed to the statement after the **END** statement of the loop. It should be understood that the statements within the **DO** group must alter some value in such a way that the loop will eventually terminate.

```
:          a = 0;              :
:    loop: DO WHILE a < 10;    :
:          c = b + c;          :
:          a = a + 1;          :
:          END loop;           :
:                              :
```

Upon normal loop termination, program execution will continue with the next executable statement following the **END** statement.

Note: In the statement '**DO WHILE x(i);** ', the usage of a programmer-defined function **x(i)** implies the relational expression (test) '**x(i) = 1**'. Hence, the statement is interpreted as

```
DO WHILE x(i) is true
```

and a branch out of loop will occur when **x(i)** is false (or 0).

```
:   DCL x Function;            :
:   DCL i BIN;                 :
:       DO WHILE x(i);         :
:       END;                   :
:   x:  PROC(i);
:       RETURN(i&1);           :
:       END;                   :
```

Iterative DO Group using a control variable

In this type of iterative **DO**, the loop is controlled by the control variables. The execution of the loop will be repeated until successive incrementing of the initial value results in a value equal to the final value. In the statement:

```
:    DO  i  =  value_1  TO  value_2  BY  value_3;    :
:                                                     :
```

The control variable **i** may not be subscripted. **value_1** and **value_2** represent the initial value and final limit, respectively, for the control variable, and **value_3** represents the increment or decrement for each iteration.

All values may be positive or negative. If the **BY** clause is omitted, it defaults to **BY +1**. If the **BY** clause is present then the **TO** clause must be present. If the value in the **BY** clause is preceded by a prefix minus operator, it implies a 'backward running' loop. The initial value would be larger than the limit value, however, the value of the **BY** clause must be positive before negation by the prefix minus, in order for meaningful code to be generated. This restriction is required in order to avoid an execution-time conflict between the generated loop control logic and a varying sign for the **BY** clause, on different invocations of the **DO** group.

Hence the sequences:

```
:    one = 1;                    :
:    DO i = 1 TO 10 BY one;      :
:                                :
```

and

```
:    one = 1;                    :
:    DO i = 10 TO 1 BY -one;     :
:                                :
```

will generate logical code, whereas the sequences:

```
:    one = -1;                   :
:    DO i = 1 TO 10 BY -one;     :
:                                :
```

and

```
:    one = -1;                   :
:    DO i = 10 TO 1 BY one;      :
:                                :
```

will not.

```
:    incr: DO i = 1 TO 10 BY 1  ::  decr: DO i = 10 TO 1 BY -1    :
:          WHILE a < b;         ::        WHILE a > b;           :
:          a = a + 1;           ::        a = a + 1;             :
:          c(i) = a + 3 * b;    ::        c(i) = a + 3 * b;      :
:          END incr;            ::        END decr;              :
:                               ::                               :
```

In the above examples, the **DO** group will be executed up to 10 times, or until the condition **a < b** becomes false. Upon normal loop termination, program execution will continue with the next executable statement following the **END** statement. In the **DO** loop named **incr**, the value of **i** after exiting the loop will be 11. Flowcharts for the above two examples follow in figure 3.6.

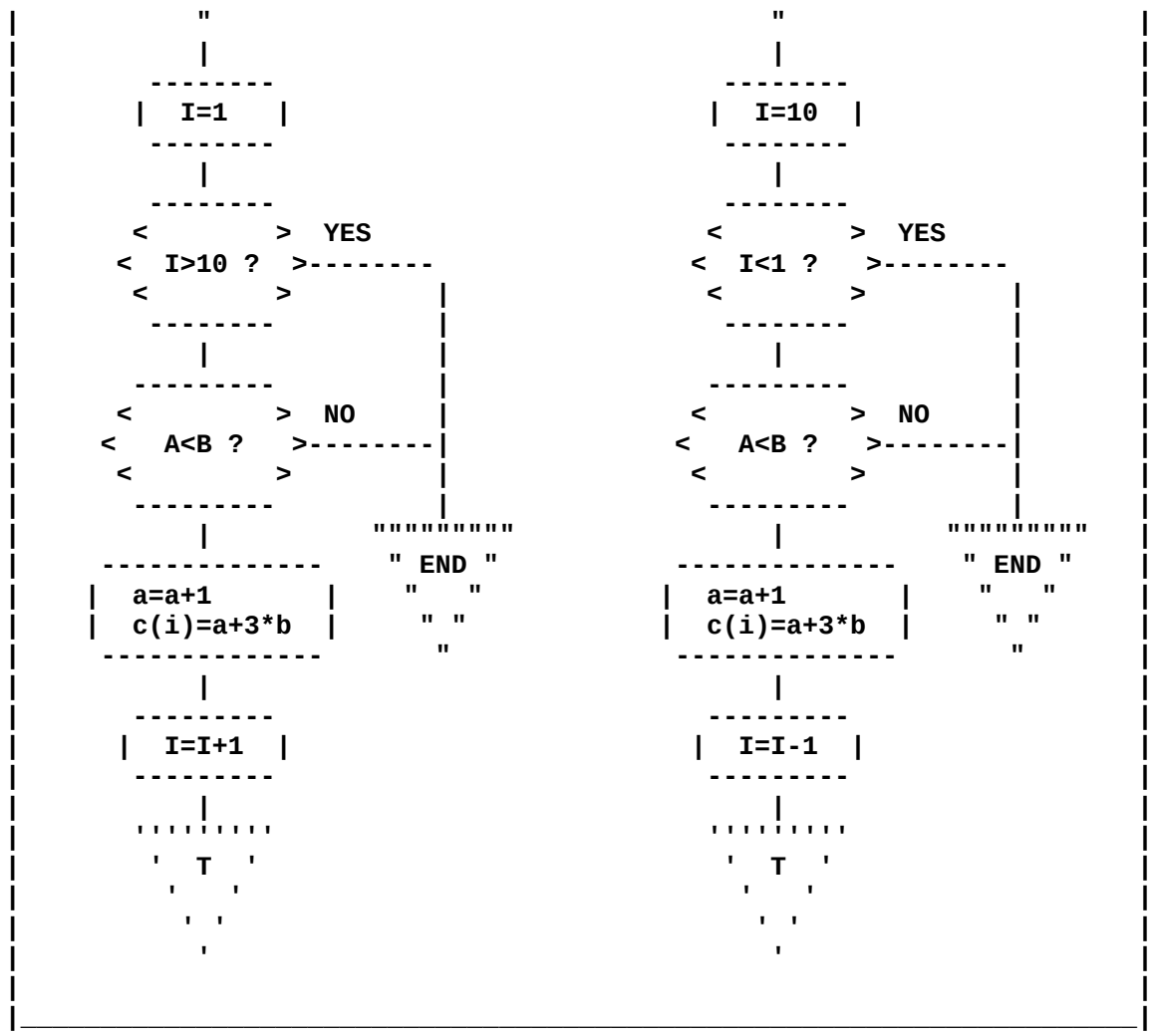


Figure 3.6 Do Loop Flow Chart

Notes:

Specifications are truncated to integer values and the sign is maintained.

In an iterative **DO**, control should not be transferred to any statement within the **DO** group from outside the **DO** group; transfer control only to the **DO** statement itself, otherwise, loop control values may not be properly set and the results will be unpredictable. Control may be transferred from within a **DO** loop to its **END** statement in order to continue the loop.

A **GOTO** statement within an iterative **DO** group can serve to abnormally terminate the **DO** loop, by specifying a label outside the **DO** group. It can also serve to prematurely advance to the next iteration of the loop by specifying a label on the corresponding **END** statement.

Any **DO** without a specific **END** statement will be 'closed' by the **END** (of the program) statement.

With regard to nested **DO** statements, the following hints are useful in improving program documentation and legibility:

A) Give labels to the DO statements
B) Use the labels as operands on corresponding **END** statements to clearly specify the range of each DO

C) Indent each DO group and use comments to explain logic.

```
:       r = mspcnr2;                                              :
:       c = mspnps2;                                              :
:   k1: DO m = 1 TO r;    /* repeat loop thru # of rows */       :
:   k2:   DO p = 1 TO c;  /* repeat loop thru # of columns */    :
:         f = f + 1;                                              :
:   k3:     DO t = 1 TO g;  /* repeat loop to # of legs */        :
:           e = e + 1;                                            :
:           pchsas(e) = mspsea1(f);                               :
:           END k3;                                               :
:         END k2;                                                 :
:       e = e + 1;                                                :
:       f = f + 1;                                                :
:       pchsas(e) = mspsea1(f);                                   :
:                         /* row ind 1-sea1(f) = ri11(f) */       :
:       e = e + 1;                                                :
:       f = f + 1;                                                :
:       pchsas(e) = mspsea1(f);       /* move row ind #2 */       :
:       END k1;                                                   :
:                                                                 :
```

NOTE: It is worth restating that there is a limit of 193 DO statements of all types allowed in a program.

IF STATEMENTS

The **IF** statement provides conditional transfer of control based on the result of relational evaluation(s). The **THEN** logic path is taken if the comparison yields true; the **ELSE** logic path is taken if the comparison yields false.

Statement Syntax:

```
IF | logical     | THEN executable    ; ( ELSE executable ;  )
   |  expression |       statement(s)   (       statement(s) )
   | relational  |                      (                    )
   |  expression |
   |             |
   -             -
```

'test' has the form:

```
| arithmetic variable   |
| arithmetic literal    |
| arithmetic constant   |
| arithmetic expression |
| logical expression    |
| relational expression |
|                       |
-                       -
```

The 'executable statement' that follows the **THEN** may be any executable statement, with the exclusion of **IF**, **PROC** and Macros with parameters.

The 'executable statement' that follows the **ELSE** may be any executable statement, with the exclusion of **PROC** and Macros with parameters.

```
:   prtctl:  lnctr =lnctr + 1;          :
:            IF lnctr > 60 THEN         :
:     page:     DO;                     :
:               lnctr = 0;              :
:               space = 7;              :
:               END page;               :
:            ELSE                       :
:     line:  space = 1;                 :
:            CALL print;                :
:                                       :
```

The print control coding above will test a line counter **lnctr** for end-of-page condition of 60 lines. If 60 lines have been printed, seven lines are spaced and **lnctr** is cleared, otherwise, 1 line is spaced.

The operators **|** (or) and **&** (and) can be used in two modes: either to separate different tests to be performed, or as normal logical operators. When used as separators, a statement consisting of all **&** or **|** operators will generate more efficient code than a statement with both. When used as logical operators, the expression(s) involved must be enclosed in parentheses. This forces the logical operation(s) to be performed before any relational test(s).

```
:   labl2: IF (a & b) > (c + 2 | d) THEN a = 10;   :
:                                                  :
```

implies only one test (greater than) to be performed.

The order of evaluation (and consequently, the result) of an expression can be changed through the use of parenthesis. The expressions enclosed in parenthesis are evaluated first to a single value before they are considered in relation to surrounding operations.

Example:

```
IF A + C = 0 & B > C | D = B - 4 then A = 10;
```

May be coded:

```
IF (A + C = 0) & (B > C) | (D = B - 4)

   then A = 10;
```

The example immediately above will yield same result as first example but increases program clarity.

The most number of logical operators allowed in an **IF** statement is sixteen.

```
:    IF ABC = 'X' |   -|            :
:       ABC = 'Y' |    |            :
:       ABC = 'Z' |    |  max of    :
:           .          |    16      :
:           .          |            :
:       ABC = '1'     -|            :
:     THEN ........                 :
```

The following 'nested-**IF**-statements' construction is allowed.

```
SYNTAX:  IF <expression> THEN <expression>;
         ELSE IF <expression> THEN <expression>;
         ELSE IF <expression> THEN DO;
              . . .
              END;
         ELSE IF  etc. . . .
         ELSE <expression>;    <--- REQUIRED]]]
```

Additional notes on nested **IF** constructs:

1) **ELSE IF. . .THEN DO**'s may not contain **IF** or **ELSE** statements. The **DO**'s may be indexed.
2) The **ELSE <expression>;** is required.
3) Final **ELSE** must close the block of nested **ELSE IF**'s.
4) If the final **ELSE** is an **ELSE DO** then it may not contain **IF** or **ELSE** statements and may not be indexed.

Example:

```
        DECLARE  1 AREC BASED(CE1CR1),
                 2  CODE     BIN(31),
                 2  ACTION   CHAR(12),
                 2  AREA     CHAR(5),
                 2  ERR      BIN(31);
        DECLARE  A   BIN(31);
        DECLARE  I   BIN(31);

  IF CODE = 1 THEN ACTION = 'ABC';

              ELSE IF CODE = 2
                      THEN ACTION = 'RETURN';

                      ELSE IF CODE = 3 THEN DO;
                                            ACTION = AREA;
                                            ENTRC INDY(ERR=#RAC);
                                            END;
                                       ELSE IF CODE = 4
                                            THEN ACTION = 'RESERVATION';
                                            ELSE GOTO ERROR1;
  A = A+CODE;
  BACKC;
ERROR1:
   ERR = 0;
   BACKC;
```

Although the programmer cannot code **IF** _____ **THEN IF**, nested **IF**'s can be accomplished by coding the nested **IF** within a non-iterative **DO**.

CHAPTER 4: EXPANDED DATA DEFINITION RULES

DATA ORGANIZATION

Data items may be coded individually, as single data elements, or they may be grouped to form arrays or structures.

Arrays

Logically related data items that have identical data attributes may be grouped to form an array. Only the array itself is given an identifier. An array is a collection of a maximum of 255 data elements. Each individual element is identified by its relative position within the array.

An array, by definition, always contains a dimension attribute and is declared by coding the attribute (enclosed in parentheses) immediately following the data identifier in the **DECLARE** statement. The dimension attribute specifies the maximum number of items in the array.

General Format:

```
 _          _
| DCL      |  ( level )  identifier  (dimension)  data          ;
| DECLARE  |  (       )
-          -

                              (0 < dimension <= 255))
```

```
:    DECLARE citycode (100) CHAR(3);    :
:    DCL labvar (10) LABEL;             :
:    DCL x (15) BIN;                    :
:                                       :
```

In the example, the first statement declares **citycode** to be an array of 100 elements, with each element a character-string 3 characters in length. The second statement declares an array of 10 elements which are label variables, each one half word in length. The third statement declares an array of 15 elements, each of which is a **BINARY** data item one half word in length.

Subscripts

Within an array, data items are accessed by the use of subscripts which specify, (within parentheses), the data item's relative position in the array. This subscript may be a literal, a variable or an expression. Subscripts must have one of the following general formats where **n** is a literal, **v** is a variable (constant), and where **v** must be of type *BIN, *BIT, *DEC or *NCS.

```
(n)
(v)
(v + n)
(v * n)
(v * n + n)
```

note: Variations are permissible, such as:

```
v - n,   n + v * n,   n + n * v,   v + n * n,
n + v,   n - v * n,   n - n * v,   v - n * n,
n - v,   n * v + n,   n * n + v,   v * n - n,
n * v,   n * v - n,   n * n - v,
```

When an array begins at a displacement of zero in storage or at a displacement less than one array entry length, and subscripts is a variable, then the subscript should be declared as *BIN.

Example:

```
:    DR95LT:  PROC;                  :
:        DCL 01 hm1rcd(6),           :
:               02 namein CHAR(20),  :
:               02 addrin CHAR(20),  :
:                     .......       :
:        DCL j  BIT(8);              :
:        DCL i  BIN;                 :
:        DO j=1 to 5;                :
:           hm1rcd(i) = hm1rcd(i+1); :
:                     .......       :
:                                   :
```

The data type of the literal must be **BINARY**. The data type of the subscript-variable may be **DECIMAL** (which will be truncated to integer), Numeric character-string (also truncated to integer), **BINARY**, or **BIT**-string. Examples of subscripted variables for each of the arrays previously declared are given:

```
:    varbl(5)                   :
:    varbl(kounter)             :
:    varbl(indx + 3)            :
:    varbl(indx - 2)            :
:    varbl(errornum * 2)        :
:    varbl(errornum * 3 + 5)    :
:    varbl(errornum * 4 - 6)    :
:                               :
```

In the preceding example, **kounter**, **indx**, and **errornum** were coded as variables within the subscript expressions. Each of these data items must have been declared with one of the data attributes acceptable for subscript variables:

```
:    DCL kounter BIN;          :
:    DCL indx PIC '99';        :
:    DCL errornum DEC (2);     :
:                              :
```

An array-to-array assignment without any element subscripts causes only the first element to be moved. References in a statement to an array without a subscript default to the first element.

It is possible to construct an array of label variables by including a dimension attribute in the **DECLARE** statement. The **GOTO** statement would then contain a subscript, specifying which element in the array of label variables is requested:

```
:    DECLARE labvar(3) LABEL;    :
:    DCL i BIN;                  :
:    labvar(1) = label1;         :
:    labvar(2) = label2;         :
:    labvar(3) = label3;         :
:              ...               :
:    label1: a = b;              :
:              ...               :
:    label2: c = d;              :
:              ...               :
:    label3: e = f;              :
:              ...               :
:    GOTO labvar(i);             :
:                                :
```

In this example, the '**GOTO labvar**' statement transfers control to the statement whose label is **label1** or **label2** or **label3** depending on the contents of **i**. **i** should be set to 1, 2 or 3 prior to the execution of the **GOTO** statement.

Structures

Structures are logical collections of named data items that need not be of the same data type or possess identical attributes, except that they must be of the same storage class. The hierarchical relationship of the elements comprising the structure is specified by the use of level numbers that are coded immediately preceding the identifier. The entire collection of data elements, called the major structure, must be coded with a level number of one. Minor structures (collections of elements contained within the major structure) are coded with level numbers greater than one. A structure, by definition, must contain a major structure. There is no such thing as a literal structure.

General Format of a Structure:

```
 _      _
| DCL    |  level  identifier  (  (dimension)  )  attributes  ;
| DECLARE |
|        |
-        -

           (0 < level <= 255)
```

```
:    DCL 1 myaddr,                  :
:          2 street CHAR(20),       :
:          2 city   CHAR(20),       :
:          2 state  CHAR(10),       :
:          2 zip    PIC '99999';    :
:                                   :
```

In the above example the major structure **myaddr** is made up of four data elements, each of which may be referenced by use of its identifier. Use of the major structure identifier in a statement would cause reference to all the elements contained within it. Structures may be assigned (only to other structures), and can be compared. In the assignment of one whole structure to another whole structure, the assignment is to the whole length of the receiving field; padding with blanks will be performed if necessary (Refer to Chapter 3, Multiple Assignment for a review of padding if necessary)

It may be to the programmer's advantage to specify smaller logical collections of data elements that are contained within the major structure. These collections, called minor structures, are given a minor structure identifier, with a level number greater than the level number of the structure in which they are contained. The elements contained within the minor structure must have level numbers greater than the minor structures.

```
:    DCL 1 employment_info,              :
:          2 personal,                   :
:            3 employee CHAR(30),        :
:            3 mapcode,                  :
:              4 street CHAR(20),        :
:              4 city   CHAR(20),        :
:              4 state  CHAR(10),        :
:              4 zip    PIC'99999',      :
:            3 title CHAR(30),           :
:          2 salary,                     :
:            3 rate         DEC(7,2),    :
:            3 pay_to_date  DEC(7,2),    :
:            3 fica_to_date DEC(7,2),    :
:            3 deductions,               :
:              4 charity DEC(5,2),       :
:              4 other   DEC(5,2);       :
:                                        :
```

In the above example the major and minor structures may be identified by the data identifiers which do not have data attributes coded in conjunction with them: (**employment_info**, **mapcode**, **salary** and **deductions** in the example above). Items on each level of a structure may be elementary data items (data items that are coded with attributes other than 'level' and 'identifier') and/or structures. The maximum number of levels is 255.

To **DECLARE** a structure and reference only a few elements in the structure, only those few elements need be detailed. This would be accomplished by using the pseudo-identifier **FILL**. **FILL** may be any data type and may have a dimension. To illustrate this technique, assume that you are receiving the following record as input:

```
:    DCL 1 aaa BASED(aaptr),         :
:          2 booking,                :
:            3 first  CHAR(10),      :
:            3 middle CHAR(10),      :
:            3 last   CHAR(10),      :
:          2 flight,                 :
:            3 number PIC'9999',     :
:            3 class  CHAR(2),       :
:          2 date,                   :
:            3 month CHAR(10),       :
:            3 day   PIC'99',        :
:            3 year  PIC'9999';      :
:                                    :
```

Suppose that in the above example only the elements **last** and **class** were to be referenced. Rather than code the **DECLARE** statement for the entire structure, as shown, the pseudo-element **FILL** could be used to **DECLARE** the non-referenced areas:

```
:    DCL 1 aaa BASED(aaptr),     :
:          2 FILL  CHAR(20),     :
:          2 last  CHAR(10),     :
:          2 FILL  CHAR(4),      :
:          2 class CHAR(2);      :
:                                :
```

No **FILL** is needed after the last essential element (**class**).

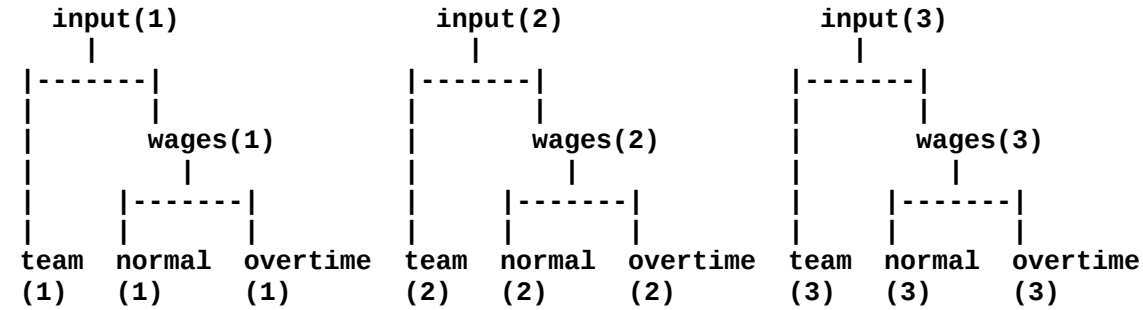
Arrays Containing Structures

An array of structures is formed by coding a dimension attribute immediately following the structure identifier:

```
:    DCL 1 input(3),                :
:          2 team CHAR(30),         :
:          2 wages,                 :
:            3 normal   DEC(5,2),   :
:            3 overtime DEC(5,2);   :
:                                   :
```

Because the major structure input has a dimension attribute coded, the elements and minor structures contained within it are arrays and must be referenced by using subscripts. This is a general rule which applies anytime a structure is arrayed. In the example above, the array 'input' and the minor structure 'team' will have the same displacement.

An in-core representation of the above example:



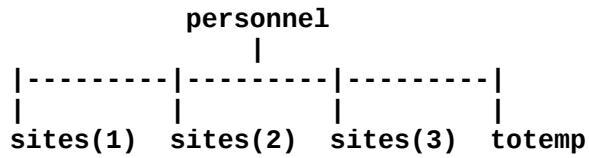
If the programmer wished to access the field **normal** he would code a subscript following it. For example: **normal(1)**, **normal(2)**, or **normal(3)**, depending upon which element he desired. All rules for subscripts, as outlined in arrays and subscripts, apply (See Subscripts, this chapter.)

Structures Containing Arrays

If an element of a structure has a dimension attribute coded in conjunction with it, the element is an array and should be referenced by the use of a subscript. A non-subscripted reference will default to a subscript of one (1).

```
:    DCL 1 personnel,               :
:          2 sites(3) CHAR(30),     :
:          2 totemp PIC'9999';      :
:                                   :
```

An in-core representation of the preceding example:



The nesting of arrays within structures is invalid, that is, only one level of each major or minor structure may have dimension attributes coded in conjunction with it.

```
:    DCL 1 personnel,              :
:          2 sites(3) CHAR(30),    :
:          2 empinfo(100),         :
:            3 rate CHAR(20),      :
:            3 empno PIC'99999';   :
:                                  :
```

The above is a valid example of a structure containing arrays because both of the arrays are on the second level. It is not possible to code a dimension attribute in conjunction with one of the level three elements and still have a valid condition:

```
:    DCL 1 personnel,                 :
:          2 sites(3) CHAR(30),       :
:          2 empinfo(100),            :
:            3 rate(7) CHAR(20),      :
:                                     :
```

The above example is invalid because **`empinfo`** and **`rate`** both have dimension attributes.

Factoring of Attributes

If several data items have identical attributes, these attributes may be factored to eliminate unnecessary coding. Factoring is specified by enclosing the identifiers of the identical data items within parentheses, separating the identifiers with commas and following this with the common data attribute(s). Given:

```
:    DCL ctr1 BIN;    :
:    DCL ctr2 BIN;    :
:                     :
```

these declarations could be factored as follows:

```
:    DCL (ctr1, ctr2) BIN;    :
:                             :
```

When factoring is performed on elements of a structure, only one level of the structure may be factored:

```
:    DCL 1 eb0eb ENTRYBLOCK,                                  :
:          2 ce1bad PTR,                                      :
:          2 ce1wka,                                          :
:            3 ebw000f,                                       :
:              4 (ebw000,ebw001,ebw002,ebw003) CHAR(1);       :
:                                                             :
```

Only one level in the above example contains factoring. The following example illustrates an invalid factoring of attributes because of an attempt to factor more than one level:

```
:    DCL 1 notes,                      :
:          2 (high,low),               :
:            3 (ones,fives,tens) BIN,  :
:            3 denominations DEC(2),   :
:          2 portraits CHAR(20);       :
:                                      :
```

Factoring of attributes is invalid for arrays, as in the example:

```
:     DCL (a,b,c) (10) BIN;   :
:                             :
```

Another method of coding **DECLARE** statements and eliminating unnecessary code is illustrated in the following examples:

```
:    DCL a BIN ALIGNED;    :
:    DCL b DEC(5,2);       :
:    DCL c CHAR(16);       :
:                          :
```

With a single **DCL** keyword, these unrelated data items could be coded more conveniently:

```
:    DCL a BIN ALIGNED,    :
:        b DEC(5,2),       :
:        c CHAR(16);       :
:                          :
```

Any number of unrelated data items might be coded in the preceding manner. Each item and its corresponding attributes is separated from the other items by the use of commas. The entire **DECLARE** statement is terminated by a semicolon.

ALIGNED and PACKED Attributes

The attributes **ALIGNED** and **PACKED** are used to align or position data. The **ALIGNED** attribute can result in more efficient code but less efficient use of space, while the **PACKED** attribute can result in more compact data allocation but less efficient code. The **ALIGNED** and **PACKED** attributes may be declared at any structure level and apply to all subdivisions (lower levels) of that structure level except where a given data element in the structure has an overriding attribute. Data items declared without the **PACKED** or **ALIGNED** attribute default to the attribute of the next higher structure level in which they appear, as just mentioned, or if they are not part of a structure they default to **PACKED**.

General Format of **ALIGNED** and **PACKED** Data:

```
 _          _
| DCL      |  identifier  data type  ( PACKED  )  ;
| DECLARE  |              attribute  ( ALIGNED )
-          -

  :   DCL  1  major            ALIGNED,     :
  :          2  maj_a  CHAR(1),             :
  :          2  maj_b  CHAR(2),             :
  :          2  min_c,                      :
  :             3  min_d  BIN  ALIGNED ;    ;
  :                                         ;
```

In the above example **min_c** and **min_d** are both aligned. Rules for alignment of **PACKED** data and **ALIGNED** data are illustrated in figure 4.1

Alignment is precedental with the following highest to lowest order:

Class of Data:	PACKED data is aligned at next:	ALIGNED data is aligned at next:
Major Structures:		
CONSTANT Storage	Byte	Byte
Other	Double Word	Double Word
Minor Structures:		
Bit	Bit	Byte
Other	Byte	(If it contains BIN(31) or PTR data): Full Word
		(If it contains BIN or LAB data): Half Word
		Else: Byte **
Array Elements:		
BIT	Byte (Overrides treatment of BIT, Elementary,)	Byte
Elementary Data:		
BIT	Bit	Byte
BIN(15) or LAB	Byte	Half Word
BIN(31) or PTR	Byte	Full Word
Other	Byte	Byte

** If the first element of a minor structure requires alignment on a half or a full word boundary then the minor structure itself is aligned on a half or full word boundary.

Figure 4.1 Packed and Aligned Data

In a structured array, bytes of padding may be added so that the size of each element will be a multiple of four if the structure contains **ALIGNED BINARY(31)** or **POINTER** data; a multiple of two if the structure contains **ALIGNED BINARY(15)** or **LABEL** data.

```
:   DCL 1 aaa (3) ALIGNED,    :
:         2 b BIN (15),       :
:         2 c CHAR (1);       :
:                             :

location      array
assigned      member
--------      ------
   0          aaa(1)
   0            b(1)
   2            c(1)
   3              -  padding
   4          aaa(2)
   4            b(2)
   6            c(2)
   7              -  padding
   8          aaa(3)
   8            b(3)
  10            c(3)
  11              -  padding
```

The padding is necessary to make each entry of array aaa an even number of bytes. This causes each **ALIGNED** item to be properly aligned in each element of the array.

In the following example **min_g** will be byte aligned:

```
:   2  minor            ALIGNED,     :
:     3  min_e  BIN,                 ;
:     3  min_f  BIT(4),              :
:     3  min_g  BIT(4);              :
:                                    :
```

If it is to be in the same byte as **min_f**, **min_g** must be **PACKED**.

```
:   3  min_g  BIT(4)  PACKED ;   :
:                                :
```

STORAGE ALLOCATION

Data items are allocated storage within the system so that they may be referenced during program execution. This storage allocation may be static, before the execution of the program, or dynamic, during execution of the program. Some allocated storage can be freed upon the explicit request of the programmer, and some upon encountering a macro containing an implicit release, e.g., ENTNC, EXITC, BACKC or ENTDC.

There are four storage classes, each of which has a corresponding keyword:

Storage class	Keyword	Allocation
AUTOMATIC	AUTO / AUTOMATIC	dynamic
BASED	BASED	static/dynamic
ENTRYBLOCK	ENTRYBLOCK	static
CONSTANT	CONSTANT	static

Data items, arrays and major structures may all have storage class attributes specified in their **DECLARE** statements, in which case the elements of all will be of the specified storage class. If a storage class is not coded in the **DECLARE** statement for a data item, array or major structure, then the storage class defaults to **AUTOMATIC**.

AUTOMATIC Storage

When a SABRETALK program is activated, a block of storage will be allocated automatically. It is this block, called the Automatic Storage Block, which provides the program with its re-entrant capability because it becomes "attached to" the ECB. All data items, arrays or major structures declared with the **AUTOMATIC** storage class attribute will reside in this core block. The core block will remain allocated as long as the program remains active. The **AUTOMATIC** storage block will be freed upon execution of a BACKC, EXITC, ENTNC or ENTDC macro. It will not be released for macros that do not cause the program block to be released, e.g., DLAYC, ENTRC or WAITC macros.

The size of the **AUTOMATIC** core block allocated will vary depending upon the requirements of the particular program segment. The Compiler will determine the program's requirements based on data items declared **AUTOMATIC** by the programmer as well as areas required by the Compiler for data conversions, etc. The Compiler will then obtain the appropriate sized block which will be 128, 381, 1055 or 4095 bytes.

General Format:

```
 _          _
| DCL      |  identifier  data type  ( AUTO      ) ;
| DECLARE  |              attribute  ( AUTOMATIC )
-          -
```

```
:    DCL inptr PTR AUTO;                                    :
:    DCL farebase DEC(5,2);   /* this  data  item  defaults  :
:                                to  AUTOMATIC  because  no  :
:                                storage   class  attribute  :
:                                is coded.              */   :
:    DECLARE labvar LABEL AUTOMATIC;                        :
:                                                           :
```

ENTRYBLOCK Storage

An Entry Control Block (ECB) is allocated to each active entry in the TPF system. In order to reference data which resides in this block, the programmer must include the **ENTRYBLOCK** storage class attribute in the data item's **DECLARE** statement. Normally the programmer does not have the need to code the **ENTRYBLOCK**.

General Format:

```
 _          _
| DCL      |   identifier   data type   ENTRYBLOCK   ;
| DECLARE  |                attribute
-          -
```

```
:    DCL 1 eboeb ENTRYBLOCK,                                    :
:          2 ce1chw PTR,                                        :
:          2 ce1bad PTR,                                        :
:          2 ce1wka,                                            :
:            3 ebw000f,                                         :
:              4 (ebw000,ebw001,ebw002,ebw003) CHAR(1),         :
:            3 ebw004f,                                         :
:              4 (ebw004,ebw005,ebw006,ebw007) CHAR(1),         :
:            3 ebw008f,                                         :
:              4 (ebw008,ebw009,ebw010,ebw011) CHAR(1),         :
:            3 ebw012f,                                         :
:              4 (ebw012,ebw013,ebw014,ebw015) CHAR(1),         :
:                                                               :
```

CONSTANT Storage

Use of the **CONSTANT** storage class attribute specifies that the data item will be allocated storage statically, prior to execution of the program, and that this storage will remain allocated within the program for the duration of program execution.

The programmer gives an identifier the **CONSTANT** storage class when the value in the field will not change. The **CONST** statement is used to specify the constant value in the field.

CONST Statements

CONSTANT storage is initialized by the use of the **CONST** statement. It is good programming practice to code the **CONST** statement immediately following the **DECLARE** statement for the data item. Each data item declared with the **CONSTANT** storage class attribute may have only one associated **CONST** statement.

Data items declared as **CONSTANTS** will reside (as will literals), in the same core block as the program.

General Format:

```
_         _
| DCL     |   identifier   data type    CONSTANT   ;
| DECLARE |                attribute
-         -

  CONST     identifier   ,   literal   ;
```

```
:    DCL error_message_1 CHAR(21) CONSTANT;                :
:    CONST error_message_1, 'invalid input message';       :
:    DCL farebase DEC(5,2) CONSTANT;                       :
:    CONST farebase, 189.95;                               :
:                                                          :
```

Literals coded in conjunction with the **CONST** statement may be of the following data types:

A) DECIMAL
B) BINARY
C) **BIT**-string
D) hexadecimal string
E) character-string
F) DECIMAL FLOAT

The literal need not be of the same data type as the data type attribute specified in the **DECLARE** statement. Always initialize the entire field specified in the **DECLARE CONSTANT** statement. With **DEC** data items use the correct number of digits in the literal of the **CONST** statement, with placement of the decimal point exactly as specified by the **DEC** in the **DECLARE** statement.

Storage for a **BIN(15) CONSTANT** statement will be changed internally to **BIN(31)** if the literal in the **CONST** statement is **BINARY**. If the programmer wishes to ensure that only two bytes are used, he should initialize the **BIN(15)** constant with a two-byte hex literal.

If a structure is declared **CONSTANT** and is initialized by several **CONST** statements at a level greater than 1, they must be initialized in logical ascending sequence or the results will be unpredictable. Structures declared as **CONSTANT** will be aligned on a full word. This is done to conserve program storage.

```
:    DCL 1 record CONSTANT,      :
:          2 team CHAR(9),       :
:          2 stamp,              :
:            3 num PIC'9999',    :
:            3 str CHAR(9);      :
:    CONST team, 'favorites';    :
:    CONST num,  '1234';         :
:    CONST str,  'sedgewick';    :
:                                :
```

Storage for **BIT CONSTANT** statement will be changed internally to a byte. If **BIT** fields are declared within a structure then the entire structure must be initialized with one **CONST** statement.

The following is incorrect:

```
:  DCL 1 K1 PACKED CONSTANT,          :
:        2 W1      BIT(1),           :
:        2 X1      BIT(1),           :
:        2 Y1      BIT(6),           :
:        2 Z1      BIN;              :
:  CONST W1,'1'B,CONST X1,'0'B;       :
:  CONST Y1,'111111'B;                :
```

The following is correct:

```
:  DCL 1 K1 PACKED CONSTANT,          :
:        2 W0,                       :
:          3 W1    BIT(1),           :
:          3 X1    BIT(1),           :
:          3 Y1    BIT(6),           :
:        2 Z1      BIN;              :
:  CONST W0,'10111111'b;              :
```

A system-equate cannot be used to initialize a field that is defined **CONSTANT**, for example, the following is illegal:

```
CONSTANT num,#pndri;
```

Constants can be arrayed, however, the entire array must be initialized in a single **CONSTANT** statement.

```
:  DCL arr(2) BIN(31) CONSTANT;        :
:  CONST arr, '0000000100000002'X ;    :
:                                      :
```

The following are all incorrect:

```
:  CONST arr, 1,2 ;                    :
:  CONST arr(1), 1 ;                   :
:  CONST arr(2), 2 ;                   :
:                                      :
```

Constants may be referenced as individual elements of an array of constants.

```
:   DCL months(12) CHAR(3) CONSTANT;                          :
:   CONST months, 'janfebmaraprmayjunjulaugsepoctnovdec';     :
:                                                             :
:   IF months(i) = inmonth THEN GOTO match;                   :
:                                                             :
```

The literal field may be no greater in size than the limit for the type of literal and only one literal per **CONST** statement is allowed.

BASED Storage

BASED storage is used primarily, but not exclusively, for data blocks that are attached to the ECB. The allocation and manipulation of TPF core blocks is controlled by the programmer through the use of the get-core, find, file, release-core, etc. type macros. It is the responsibility of the programmer to initialize the pointer for **BASED** storage so that it contains the address of the data block he wishes to reference after storage allocation has been accomplished. One method of initializing the pointer with the address of the storage area is to obtain it from the core block reference word of the Entry Control Block (ECB) by use of an Assignment statement. Note that the pointer name in the **BASED** attribute is implicitly declared, i.e., the programmer should not code a separate declare for the pointer.

General Format:

```
_          _
| DCL      |   identifier   data type   BASED   (pointer)   ;
| DECLARE  |                attribute
-          -
```

```
:    DECLARE 1 input (5) BASED (pointer1),    :
:              2 shop CHAR(30),               :
:              2 roster CHAR(50),             :
:              2 phone CHAR(8),               :
:              2 data CHAR(123);              :
:    GETCC d1,l2;                             :
:    pointer1 = ce1cr1;                       :
:                                             :
```

After initializing the pointer, as illustrated, all references to data items based on this pointer will be qualified by the address the pointer contains. In other words, the compiler will generate instructions with the pointer address as base, and use the relative location of the data item in the structure as displacement. After pointer initialization, the data items in the structure input may be referenced.

```
:   shop = stringl;   :
:   .                 :
:   RELCC d1;         :
```

```
:                    :
```

Once the block of storage has been released, for example, by the use of a release-core type macro, data items in the released block must not be referenced. The programmer may request a new core block, not necessarily on the same data level, and again initialize the pointer. After this has been accomplished, the **BASED** data items, now residing in the new core block, may be referenced. Another method is to declare the core address reference word as the pointer to the structure. The compiler recognizes this pointer as ECB based and will produce the necessary code to initialize the pointer.

Explicit Pointer Usage

Pointers are variables containing addresses that refer to data items. The data attribute **POINTER** or **PTR** is used to **DECLARE** a pointer explicitly as follows:

General Format:

```
_          _              _          _
| DCL      |   identifier   | PTR      |  ;
| DECLARE  |                | POINTER  |
-          -                -          -

:    DECLARE recptr POINTER;     :
:    DCL input PTR;              :
:                                :
```

One full word of storage is allocated in **AUTOMATIC** storage for the pointer.

When pointers are declared in this manner, data items will not be automatically based upon them. The programmer must use the pointer qualification composite (**->**) to specify which data item should be based on the pointer. The Compiler will then generate the instructions to complete the required operation, using the pointer specified as the base for the data item and the relative location of the data item as the displacement. A pointer used to qualify a reference to a data item must reside in **AUTOMATIC** storage.

```
:    DCL outptr PTR;              :
:    DCL 1 input BASED(inptr),    :
:         2 cost CHAR(30),        :
:         2 home CHAR(30),        :
:         2 file CHAR(68);        :
:    GETCC d6,l0;                 :
:    outptr = ce1cr6;             :
:    outptr -> home = home;       :
:                                 :
```

In the preceding example, the last statement illustrates both explicit and implicit pointer qualification. On the right hand side of the Assignment statement the data item **home** is referenced in the implicit manner. The Compiler will generate instructions using the address contained in **inptr** as the base for this reference and the displacement will be determined by the location of **home** in the structure, (base + 30). The receiving field of the assignment specifies explicit pointer qualification, causing the Compiler to override the normal base of **home** and use the address contained in **outptr** as the base for this reference. Displacement from this base is again determined by the location of **home** in the structure.

General Rules for the Use of Pointers:

1) A pointer can be initialized to contain an address in the following ways:

   a) By use of a pointer-to-pointer Assignment statement. (e.g., **`outptr = ce1cr3;`**)
   b) By use of a **START** statement or macro in which is specified that the contents of a register be stored into the pointer.
   c) By use of the **ADDR** built-in function, which can be used to obtain the address of a data item, array, structure, or an element within an array or structure (e.g., **`aptr = ADDR(data);`**)
   d)The pointer being an input parameter in a programmer-declared function or an internal procedure.

2) Pointer qualifiers may not be subscripted, as in:
```
x = rptr(i) -> z;
```

3) Pointers may not be further qualified by other pointers, as in:
```
a = rptr -> aptr -> b;
```

4) Pointers may be considered as address constants and may have **BINARY** literals or variables added to or subtracted from them. The result of such an addition or subtraction may be assigned to a pointer, as in:
```
strptr = strptr + #itemln;
```

5) Data based on a pointer must not be referenced until the pointer is properly initialized, except in the case of ECB based pointers.

6) Once the storage allocated to a **BASED** data item has been released, that data item must not be referenced.

7) Whenever a pointer is received as a parameter in a macro, or in a procedure, it should be tested to insure that it is not zero.

8) A maximum of 92 pointers, implicitly or explicitly declared, may be used in the program.

DEFINED ATTRIBUTE AND DEFINED STORAGE

The **DEFINED** attribute allows the programmer to **DECLARE** a data item that will occupy all or part of the storage allocated to another data item called the base-identifier. The declaration is not a method of allocating storage, rather it is a way of re-defining existing storage so that the storage class remains the same but the attributes and/or identifiers may change. The specification of a base-identifier in the **DEFINED** attribute is mandatory. The starting address of the **DEFINED** data item will be the same as the starting address of any other data item whose name is used as the **DEFINED** base-identifier.

General Format:

```
 _          _                 _          _
| DCL      |   identifier   data type   | DEF      |   base-       ;
| DECLARE  |                attribute   | DEFINED  |   identifier
-          -                            -          -

      :   DCL flightcs CHAR(4);                    :
      :   DCL fltncs PIC'9999' DEFINED flightcs;   :
      :                                            :
```

The programmer can use this attribute to treat a given quantity as two different data types (in the example, as character-string and Numeric character-string), or to reuse an area of storage in a different manner at some later time. In the above example, the actual data in the field would remain the same, while the Assembler Language instructions generated would be for a character-string field or a Numeric character-string field, depending upon the identifier used. The **DEFINED** data item **fltncs** will have the same starting address and will occupy the same area as the base-identifier, **flightcs**. When these four bytes are referred to by the identifier **fltncs**, they will be treated as a Numeric character-string field. If the identifier **flightcs** is used, the field will be treated as if it contained character-string data.

The practicality of defining a field in this manner becomes evident when the programmer compares or assigns zoned numeric literals to the field. Character-string literals, such as '1234' (which contain numbers), may only be compared or assigned to fields declared as character-strings, but character-string fields may not be coded in arithmetic expressions. Using the preceding **DECLARE** statements the programmer could compare or assign a character-string literal to **flightcs** and then use the identifier **fltncs** in an arithmetic expression.

```
:   IF flightcs = '1234' THEN fltncs = fltncs + 1.;   :
:                        ELSE flightcs = '0000';      :
:                                                     :
```

Note: The preceding situation also could be handled through the use of the **NSTR** built-in function which is covered in Chapter 5, **NSTR**

The following example illustrates use of an area of storage in a different manner at a later time.

```
:   DCL 1 input BASED(inptr),     :
:         2 cheese CHAR(25),      :
:         2 age PIC'999';         :
:   DCL rate DEC(5,2) DEF age;    :
:   rate = 100.25;                :
:                                 :
```

Originally, **input** contains whatever values were assigned to **cheese** and **age**. Later, as a result of the assignment of a value to **rate** (which occupies the same area of storage as **age**), **input** contains the values in **cheese** and **rate**, with **rate** occupying (and therefore destroying the previous contents of) the same storage area that **age** occupied.

General Rules for **DEFINED** Storage:

1) The attributes of the **DEFINED** data item may not be inconsistent with the attributes allowable to the storage class of the data item whose name is used as the base-identifier. For example, if the base identifier belongs to the **CONSTANT** class of storage, then a data item that is a DEFINED variable would represent an inconsistency.

2) The length of the data item should not exceed the length of the base-identifier unless the storage class of the fields is **ENTRYBLOCK** or **BASED**. When the length is greater than the base-identifier, the field immediately following the base-identifier will be overlaid if an assignment is made to the data item.

```
:    DCL 1 input BASED(a1),          :
:          2 empnum PIC'99999',      :
:          2 rate DEC(5,2);          :
:    DCL part CHAR(7) DEF empnum;    :
:    part = 'ABCDEF' ;               :
:                                    :
```

In the above example, **part** will be allocated the same five bytes allocated **empnum** plus the first two bytes allocated to **rate**. The assignment of a value to **part** will alter all of **empnum** and the first two bytes of **rate**.

3) If the **DEFINED** is included in the original **DECLARE** statement for a structure, the **DEFINED** items/structure must be at the bottom of the structure. This is because the compiler cannot specifically recognize the end of a **DEFINED** structure and therefore continues to treat any declarative statements that follow the **DEFINED**, as part of the **DEFINED**.

```
:    DCL 1 rec,                        :
:          2 post CHAR(12),            :
:          2 box,                      :
:            3 addrs CHAR(20),         :
:            3 phone PIC'9999999',     :
:          2 misc DEFINED addrs ,      :
:            3 sub1 CHAR(10) ,         :
:            3 sub2 CHAR(10) ;         :
:                                      :
```

Any number of **DEFINED** statements can be coded in sequence, at the end of the structure.

4) (See restrictions of **DEFINED** items in **%INCLUDEAF** files, this chapter)

INCLUSION STATEMENTS

%INCLUDEAF Statement

The **%INCLUDEAF** statement is a statement which gives the programmer the ability to retrieve (non-executable) **DECLARE** statements in pre-compiled form from the **%INCLUDEAF** library and have them inserted into his program.

When the **%INCLUDEAF** statement is encountered during compilation, the precompiled declares are retrieved from the **%INCLUDEAF** library and only those declares referenced by the program are made known to the program. This has the effect of decreasing compile time and saving core requirements. (See Chapter 7 **ICAFNO/YES OPTIONS** for further information)

General Format:

```
%INCLUDEAF   membername   ( ,membername   (,...) )  ;

:    %INCLUDEAF MI0MI;           :
:    %INCLUDEAF EB0EB, WA0AA;    :
```

MI0MI, **EB0EB** and **WA0AA** would reflect the identifiers of particular sets of declarations on file.

General Rules for **%INCLUDEAF** Statements:

1) Details for loading members to the **%INCLUDEAF** library are found in the SABRETALK Installation and Maintenance Guide.

2) Only (non-executable) **DECLARE** statements may be inserted in one's program by use of **%INCLUDEAF**.

3) **DECLARE** statements in a **%INCLUDEAF** member may not have a storage class attribute of **AUTOMATIC** or **CONSTANT**.

4) If more than one set of **%INCLUDEAF** library members is requested, the member identifiers can be coded in one **%INCLUDEAF** statement, separated by commas. The maximum number of **%INCLUDEAF** members that can be requested by a given program is 24.

5) **%INCLUDEAF** members may contain more than one structure if the structure does not contain **DEFINED** items. Any structure containing **DEFINED** items must be placed in a separate **%INCLUDEAF** member.

6) If a specified **%INCLUDEAF** member is not in the **%INCLUDEAF** library, the compiler will not search the **%INCLUDE** library.

Since the Entry Control Block (ECB) is the primary control medium in a TPF environment, its description is usually found catalogued in the **%INCLUDE** and/or **%INCLUDEAF** libraries. In addition, the primary application data records in a given TPF system are usually found in the **%INCLUDEAF** and/or the **%INCLUDE** libraries.

%INCLUDE Statement

The **%INCLUDE** statement is a statement which gives the programmer the ability to retrieve source statements from the **%INCLUDE** library and have them inserted into his program at the point where the statement is coded. These source statements may be executable statements or (non-executable) **DECLARE** statements. (See Chapter 7 **ICAFNO/YES OPTIONS** for further information)

General Format:

```
%INCLUDE   membername   ( ,membername   (,...) )  ;
                        (               (    ) )
                        (                    )

:    %INCLUDE error;          :
:    %INCLUDE eb0eb, wa0aa;   :
```

General Rules for **%INCLUDE** Statements:

1) **%INCLUDE** members may not contain **%INCLUDEAF** statements or **%INCLUDE** statements.

2) Details for loading members to the **%INCLUDE** library are found in the SABRETALK Installation and Maintenance Guide.

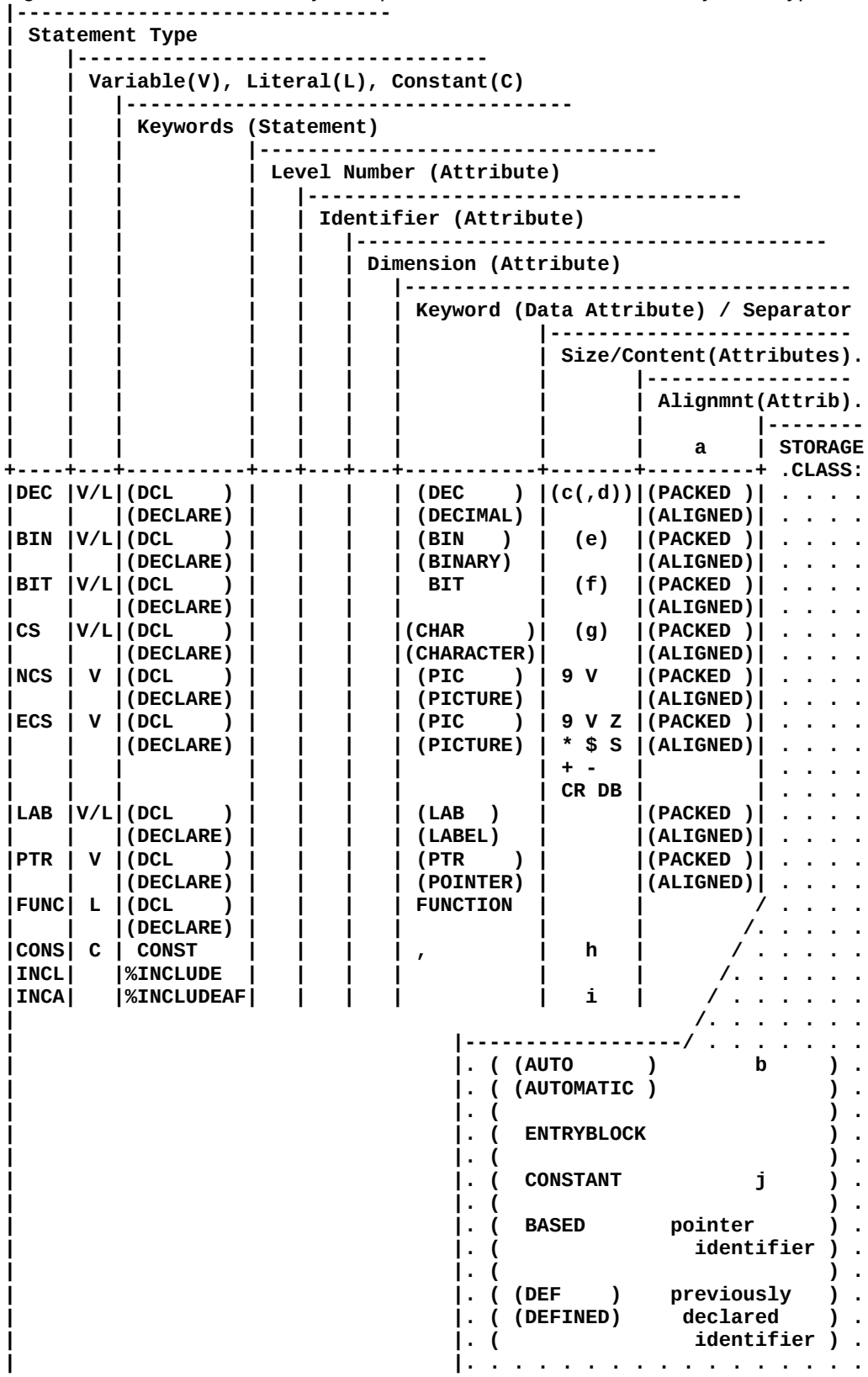
3) If an internal **PROCEDURE** is to be inserted by use of the **%INCLUDE** statement, the **%INCLUDE** statement should be coded outside the logical flow of the program.

4) If more than one set of **%INCLUDE** library members is requested, the member names, separated by commas, can be coded in one **%INCLUDE** statement.

NOTE: If member names are coded as part of one statement, be aware that if a member is not found on file, then subsequent member(s) will not be searched for on file.

DATA STATEMENT STRUCTURE SUMMARY

Figure 4.2 illustrates a summary of Explicit Data Statement Structure by Data Type



Key:

a) **PACKED is an alignment default.**

b) **AUTOMATIC is a storage default.**

```
c)  The parentheses around 'c' are necessary,
    Number of decimal digits:      Max = 15        Min = 1
d)  The parentheses around 'd' (and 'd') are optional,
    Position of decimal point from low order digit:
                Default = 0        Max = 15        Min = 0
e)  15 or 31 :   Default = 15
f)  Number of bits:                Max = 32
g)  Number of characters:          Max = 4087
h)  Literal of type:
                    Decimal                9.3
                    Binary                 189
                    Bit-string            '010'B
                    Character-string      'M N'
                    Hexadecimal-String    'A3F'X
i)  %INCLUDEAF may only be used with:
              ENTRYBLOCK
              BASED
j)  Number of constant characters: Max = 256
```

Figure 4.2 Explicit Data Statement Structure By Data Type.

CHAPTER 5: EXPANDED EXECUTABLE STATEMENT RULES

PRECISION

The following rules specify the expected precision (p) and scale (q) that will result from arithmetic operations. The notations are uniquely defined for use in the arithmetic algorithms that follow:

p	represents the total number of digits in the result.
q	represents the number of fractional digits in the result.
p1	represents the total number of digits in the first operand.
q1	represents the number of fractional digits in the first operand.
p2	represents the total number of digits in the second operand.
q2	represents the number of fractional digits in the second operand.

The 'result' of an arithmetic operation refers to the value obtained after an operation has been completed. If this value is to be further operated on in the same statement, it is termed an 'intermediate result'.

```
:   g = b + c;   :
:                :
```

The result of **b+c** is a final result that will be assigned to **g**.

```
:   g = b + c + d;   :
:                    :
```

The result of **b+c** is now an intermediate result that will be added to **d**, giving a final result that will be assigned to **g**.

Decimal Arithmetic

The following rules pertain to **DECIMAL** arithmetic operations on operands, which have been declared or converted to **DEC(p,q)**. If one of the operands is **BINARY**, conversion is as follows: **BIN(15)** to **DEC(5,0)**, **BIN(31)** to **DEC(11,0)**. The largest allowable **DECIMAL** precision is fifteen (15) digits.

Arithmetic Operations

The following formulas are used to determine the precision and scale of the intermediate result of arithmetic operations:

Addition and Subtraction

```
p = 1 + max (p1 - q1, p2 - q2) + max (q1, q2)
q = max (q1, q2)

:   DCL p5v4 DEC(5,4);   :
:       p7v3 DEC(7,3);   :
:                        :
```

In the expression **p5v4 + p7v3** the intermediate result will have a precision (p,q) of (9,4):

```
p = 1 + 4 + 4 = 9
q = 4
```

Multiplication

```
p = p1 + p2 + 1
q = q1 + q2
```

In the expression **p5v4 * p7v3** the intermediate result will have a precision (p,q) of (13,7):

```
p = 5 + 7 + 1 = 13
q = 4 + 3 = 7
```

Division

```
p = 15
q = 15 - ( (p1 - q1) + q2)
```

In the expression **p5v4 / p7v3** the intermediate result will have a precision (p,q) of (15,11):

```
p = 15
q = 15 - ( (5 - 4) + 3) = 11
```

Precision of **MOD** Operations on Decimal Data

The **MOD** function, frequently termed a 'remainder after division' has its own formula for computing precision. After a **MOD** operation the precision (p,q) is:

```
p = MIN (15, p2 - q2 + MAX (q1, q2) )
q = MAX (q1, q2)
```

In the expression **MOD(p5v4, p7v3** , the intermediate result will have a precision (p,q) of (8,4):

```
p = MIN (15, (7 - 3 + MAX (4, 3) ) = 8
q = MAX (4, 3) = 4
```

Results of Arithmetic Operations on **BINARY** Numbers

All **BINARY** arithmetic operations involve the use of a full word, 32 bits, regardless of the declared size of the operands - **BIN(15)** - such that all intermediate results are **BIN(31)**.

Truncation

Arithmetic operations take place only after all necessary conversions have been performed. Since the maximum size of a **DECIMAL** field is fifteen digits, truncation is necessary if this limit is exceeded. Digits truncated may or may not be significant digits. The method of truncation employed is based upon whether the result is intermediate or final. When truncation is associated with a final result, the result is aligned on the decimal point of the receiving field and excess high order and low order digits are lost. With regard to intermediate results, truncation may occur after an intermediate result has been obtained and before the next operation takes place. Truncation of low order fractional digits occurs first. While truncation usually affects low order fractional digits, high order digits may be truncated if the size of the intermediate result exceeds the largest allowable precision for the data type.

Consider the following multiplication of four numbers, three of which are **DEC(5,2)** and a fourth which is **DEC(7,2)**. The receiving field is **DEC(15,0)**:

```
(5,2) * (5,2) * (5,2) * (7,2)
```

The rules for determining the size of each intermediate result are those for **DECIMAL** multiplication.

Stage 1: **(5,2) * (5,2) yields (11,4)**

The intermediate result obtained at this stage falls within the limit of the 15 **DECIMAL** digits.

Stage 2: **(11,4) * (5,2) yields (17,6)**

In order to decrease the size of this intermediate result to within the maximum allowable, two of the low order fractional digits are truncated to yield a result whose specification is (15,4).

Stage 3: **(15,4) * (7,2) yields (23,6)**

Truncation of the final result is achieved by aligning the decimal point of the final result (23,6) with that of the receiving field (15,0). Therefore, the 2 high order digits and the six low order fractional digits are truncated. This could mean a loss of significance, depending upon the actual values originally in the fields.

FUNCTIONS

There are two types of functions: Built-in functions and Programmer-declared functions. Built-in functions are a part of the compiler; the programmer writes programmer-declared functions. Both types are invoked with statements of similar format. The parameters being passed are coded immediately following the function label, separated by commas and enclosed in parentheses. No function invocations, other than the pseudo-variable built-in functions (**BSTR**, **CSTR** and **NSTR**) are allowed as parameter assignments in TPF macro statements.

Built-in functions are a set of pre-defined functions that are an intrinsic part of SABRETALK. These include:

(1) Arithmetic functions which return characteristics about arithmetic data,
(2) string functions which return characteristics about a string or some shifted version of a string or which allow some attributes of the string to be temporarily over-ridden,
(3) an address function which returns an address,
(4) a case function which produces conditional branches.

Built-in functions are not to be declared. The built-in function names are SABRETALK keywords, and may not be used by programmers as identifiers or labels.

Programmer-declared functions are sets of statements, invoked by the use of the function label, the label usually used as part of a sending field or receiving field in an assignment statement. Programmer-declared functions operate on the data passed to them and return a value to the invoking statement.

Built-in Functions

The pseudo-variable built-in functions **BSTR**, **CSTR,** and **NSTR** are the only functions that may be coded as the receiving field of an assignment statement or in the register loading/storing portions of a macro or **START** statement.

A list of the built-in functions, by categories, follows:

Arithmetic

Label	Function
ABS	absolute value
MAX	maximum
MIN	minimum
MOD	modulo
SIGN	algebraic sign
ROUND	rounds decimal data

String

Label	Function
ALPHA	alphabetic scan
NUMERIC	non-alphabetic scan
INDEX	indexed scan
SHL	shift left
SHR	shift right
BSTR	bit-string
CSTR	character-string
NSTR	numeric character-string
VSTR	variable-length string
LSTR	find length of a data field

Address

Name	Function
ADDR	address

Case

Name	Function
CASE	case

Arithmetic functions return results that depend upon the precision of the parameters. If a parameter is an expression, the precision of the result of the expression is used.

ABS Built-in Function

```
ABS (x)
```

The above returns the absolute value of **x**.

General Rules for **ABS** Built-in Functions:

1) **x** is a required parameter, which must be an arithmetic, logical or relational expression.

2) **x** may be subscripted and/or pointer qualified:

```
:   DCL num BIN(31),   :
:       val BIN;       :
:   num = -13;         :
:   val = ABS (num);   :
:                      :
```

Result of the above: val = +13.

MAX Built-in Function

```
MAX   (x,y   (,p   (,...) )   )
             (     (    ) )
             (            )
```

The above returns the value of whichever parameter contains the greatest value.

General Rules for **MAX** Built-in Functions:

1) The data types may be dissimilar in which case conversions will be performed.

2) **x** and **y** are required parameters which must be arithmetic, logical or relational expressions.

3) **x** and **y** may be subscripted and/or pointer qualified.

4) **p** is an optional parameter or optional parameters (there may be any number of parameters) which must be arithmetic, logical or relational expression.

5) **p** may be subscripted and/or pointer qualified.

6) Precision of result returned will be (n,m), where m is obtained from the parameter with the most fractional digits. n is equal to k+m where k is obtained from the parameter with the most integer digits. The precision (n,m) would be large enough to contain all possible significant digits in the result.

```
:   DCL day_rate   DEC(7,5),                        :
:       month_rate DEC(3,1),                        :
:       trial_rate DEC(9,4),                        :
:       val        DEC(11,5);                       :
:   day_rate   = 99.9;                              :
:   month_rate =   .3;                              :
:   trial_rate = 98.;                               :
:   val = MAX (day_rate,month_rate,trial_rate);     :
:                                                   :
```

val, above, will become 99.9, precision returned is (10,5).

MIN Built-in Function

```
MIN   (x,y   (,p   (,...) )   )
             (     (    ) )
             (            )
```

The above returns the value of whichever parameter contains the least value.

General Rules for the **MIN** Built-in Function:

1) The data types may be dissimilar in which case conversions will be performed.

2) **x** and **y** are required parameters which must be arithmetic, logical or relational expressions.

3) **x** and **y** may be subscripted and/or pointer qualified.

4) **p** is an optional parameter or optional parameters (there may be any number of parameters) which must be arithmetic, logical or relational expression.

5) **p** may be subscripted and/or pointer qualified.

6) Precision of result returned will be (n,m), where m is obtained from the parameter with the most fractional digits. n is equal to k+m where k is obtained from the parameter with the most integer digits. The precision (n,m) would be large enough to contain all possible significant digits in the result.

```
:   DCL new_price   DEC(11,1),                          :
:       old_price   DEC(5,5),                           :
:       quote_price DEC(1),                             :
:       val DEC(15,5);                                  :
:   new_price  = 1.3;                                   :
:   old_price  =  .99;                                  :
:   quote_price= 9.;                                    :
:   val = MIN (new_price, old_price, quote_price);      :
:                                                       :
```

val, above, will become .99, precision returned is (15,5).

MOD Built-in Function

```
MOD (x,y)
```

The above returns the remainder from the division of **x** by **y**.

General Rules for the **MOD** Built-in Function:

1) **x** and **y** are required parameters which must be arithmetic, logical or relational expressions.

2) **x** and **y** may be subscripted and/or pointer qualified.

3) Fractional digits of the remainder will be truncated.

4) If the data types of the parameters are dissimilar, a conversion will be performed according to the rules for conversions in arithmetic expressions.

```
:   DCL (bundle,newspaper) BIN(31);   :
:   DCL extra BIN;                    :
:   bundle = 47;                      :
:   newspaper = 5;                    :
:   extra = MOD (bundle,newspaper);   :
:                                     :
```

The above will return, in extra, the value 2 as a remainder.

ROUND Built-in Function

```
ROUND(X,Y);
```

This function rounds the **DECIMAL** value **X** to the **Y**th decimal place by adding 5 to the **Y**+1 decimal place and truncating the result to **Y** decimal places.

General Rules for the **ROUND** Built-in Function:

1) The **ROUND** function may be used anywhere a **DECIMAL** data type may be used.

2) **X** may be a **DECIMAL** variable, literal, constant, or any arithmetic expression whose final result is **DECIMAL**.

3) **Y** is the place value to round to. It must be a **BINARY** literal or constant.

4) **Y** must be 0 or greater. A value of 0 causes the **DECIMAL** to be rounded to an Integer.

5) If **X** does not contain **Y**+1 decimal places a warning will be issued, since no actual rounding can occur.

```
:                                          :
:   DCL VARIABLE      DEC(7,5);            :
:   DCL ANSWER        DEC(5,2);            :
:   VARIABLE = 23.56789;                   :
:   ANSWER = ROUND(VARIABLE * 0.0625,2)    :
:                                          :
```

In the above, **ANSWER** will have the result of the product of **VARIABLE * 0.625** rounded to two (2) decimal places.

Examples:

```
DECLARE  A       DEC(11,5);
DECLARE  B       DEC(11,4);
DECLARE  C       BIN(31) CONSTANT;
CONST    C,      3;

1)       A = ROUND(B,2);
                        MVO   $TEMPDBL(6,R7),B$(5,R7)
                        MVN   $TEMPDBL+5(1,R7),B$+5(R7)
                        AP    $TEMPDBL(6,R7),=XL001'5C'
                        ZAP   $TEMP001(16,R7),$TEMPDBL(6,R7)
                        MP    $TEMP001(16,R7),=XL002'100C'
                        ZAP   A$(6,R7),$TEMP001+10(6,R7)
```

```
2)        B = ROUND((A*0.625+B)/2,C);
                          ZAP   $TEMP001(9,R7),=XL003'00625F'
                          MP    $TEMP001(9,R7),A$(6,R7)
                          MVC   $TEMPDBL(8,R7),$TEMP001(R7)
                          MVN   $TEMPDBL+7(1,R7),$TEMP001+8(R7)
                          ZAP   $TEMP002(9,R7),B$(6,R7)
                          MP    $TEMP002(9,R7),=XL002'100C'
                          AP    $TEMP002(9,R7),$TEMPDBL(8,R7)
                          MVC   $TEMP003(8,R7),$TEMP002(R7)
                          MVN   $TEMP003+7(1,R7),$TEMP002+8(R7)
                          LA    R1,0002
                          CVD   R1,$TEMP004(R7)
                          ZAP   $TEMP005(16,R7),$TEMP003(8,R7)
                          DP    $TEMP005(16,R7),$TEMP004+2(6,R7)
                          MVC   $TEMP006(8,R7),$TEMP005+2(R7)
                          MVC   $TEMP007(8,R7),$TEMP006(R7)
                          AP    $TEMP007(8,R7),=XL001'5C'
                          MVC   B$(6,R7),$TEMP007+2(R7)
```

SIGN Built-in Function

SIGN (x)

This function analyzes **x** and determines its algebraic sign. If **x** is greater than zero, the value returned is a positive binary one. If **x** is zero, zero is returned and if **x** is less than zero, a negative binary one is returned.

General Rules for the **SIGN** Built-in Function:

1) **x** is a required parameter that must be an arithmetic, logical or relational expression.

2) **x** may be subscripted and/or pointer qualified.

3) **x** may be of any arithmetic data type (NCS, **DEC**, **BIN**) except **BIT**.

```
:   DCL trial_balance DEC(3,2),     :
:       audit BIN;                  :
:   trial_balance = -7.43;          :
:   audit = SIGN (trial_balance);   :
```

In the above, audit becomes a minus one (-1).

ALPHA Built-in Function

ALPHA (a)

The **ALPHA** built-in function is defined by the installation, usually to mean 'find the location of the first alphabetic character'. The definition of the function is predicated upon the characteristics of the table accessed by the function. The name of the table accessed by the compiler is a compiler option (See Chapter 7, for additional compiler options information.)

The function scans left to right the string of characters provided as the parameter. It returns a binary value equal to the position of the first character encountered (for which the accessed table indicated "stop scan"). If the first character in the string meets the criteria, a binary one (1) is returned. If the scan finds no character meeting the criteria, a zero (0) is returned.

General Rules for the **ALPHA** Built-in Function:

1) **a** is a required parameter which may not be an expression, meaning it can not contain an operator.

2) **a** may be subscripted and/or pointer qualified.

3) **a** is treated as a "string" of characters, i.e., a single character or a series of consecutive characters.

In the following example, assume that the table associated with the **ALPHA** function is set up to 'stop scan' on alphabetic characters (A through Z).

```
:   DCL list CHAR(30),        :
:       val BIN;              :
:   list = '9300 nw 36 st';   :
:   val = ALPHA (list);       :
:                             :
```

In the above, **val** becomes a **BINARY** 6.

NUMERIC Built-in Function

```
NUMERIC (a)
```

The **NUMERIC** built-in function is defined by the installation, usually to mean 'find the location of the first non-alphabetic character'. The definition of the function is predicated upon the characteristics of the table accessed by the function. The name of the table accessed by the compiler is a compiler option (See Chapter 7, for additional compiler options information.)

The function scans left to right the string of characters provided as the parameter. It returns a binary value equal to the position of the first character encountered (for which the accessed table indicated "stop scan"). If the first character in the string meets the criteria, a binary one (1) is returned. If the scan finds no character meeting the criteria, a zero (0) is returned.

General Rules for the **NUMERIC** Built-in Function:

1) **a** is a required parameter which may not be an expression.

2) **a** may be subscripted and/or pointer qualified.

3) **a** is treated as a "string" of characters.

In the following example assume that the table associated with the **NUMERIC** function is set up to 'stop scan' on non-alphabetic characters (not A through Z).

```
:   DCL cost CHAR(12),        :
:       val BIN;              :
:   cost = 'pending test';    :
:   val = NUMERIC (cost);     :
:                             :
```

In the above, **val** becomes a **BINARY** 8.

INDEX Built-in function

```
INDEX   (a,b   (,o)  )
               (  )
```

The **INDEX** function causes a search of a specified "string" (**a**) for a specified "string" (**b**). **b** is compared to the first n characters of **a**, where n is the length of **b**. The comparison is repeated every **o** characters until a match is found or the end of string **a** is encountered. If the **b** configuration is found in **a**, the starting location of string **b** within string a is returned. If **b** is not contained within **a**, then the value returned is a binary zero (0). **o** has a maximum of 69.

General Rules for the **INDEX** Built-in Function:

1) **a** and **b** are required parameters which must be one of the following: a variable, a **BSTR**, a **CSTR**, an **NSTR**, a subscripted variable, a structure, a character-string literal or a **BIT**-string. If **BSTR**, **CSTR** or **NSTR** built-in functions are used, the 3rd parameter of the **BSTR**, **CSTR** or **NSTR** must be a literal. (See number 5.)

2) If **a** or **b** is a **BIT**-string, it must be byte aligned and a byte multiple in length.

3) **a** and **b** may be pointer qualified.

4) **a** and **b** are treated as "strings" of characters.

5) **b** must be of shorter length (not equal or greater) than **a**. In order to be sure that the lengths are correct, if **BSTR**, **CSTR** or **NSTR** are used, the length parameter (3rd parameter) of the **BSTR**, **CSTR** or **NSTR** must be a literal,

6) **o** is an optional parameter which, if coded, must be a **BINARY** literal.

7) If **o** is not coded, it defaults to one (1).

```
:   DCL text CHAR(10),           :
:       key CHAR(2),             :
:       val BIN;                 :
:   text = 'abcgopmlzd';         :
:   key  = 'pm';                 :
:   val  = INDEX (text,key);     :
:   val  = INDEX (text,key,2);   :
:                                :
```

The above statement: **val = INDEX (text,key);** would result in **val** becoming 6.

The above statement: **val = INDEX (text,key,2);** would result in **val** becoming 0.

```
:   DCL 1 rcd,                          :
:         2 odd (2) CHAR (200),         :
:         2 even (2) CHAR(200);         :
:   DCL match (3) CHAR(200);            :
:   DCL flg       BIN (31);             :
:   flg = INDEX (rcd,match(2),60) ;     :
:                                       :
```

In the above example a structure is searched to find a match with a subscripted variable. Any comparison of the second element of **match** would be to the first element of **odd** or to the first element of **even**.

SHL Built-in Function

```
SHL (x, y)
```

The **x** parameter is converted to a 32 bit **BIT**-string, if necessary, and is logically shifted **y** bits to the left. There is no arithmetic evaluation of logically shifted **x**. Zeroes are introduced in the vacated bit positions on the right.

General Rules for the **SHL** Built-in Function:

1) **x** and **y** are required parameters which must be arithmetic, logical or relational expressions.

2) **x** and **y** may be subscripted and/or pointer qualified.

3) If, when converted to **BIT**-string, **x** is less than 32 bits in length, zero bits are added on the left. **x** may be a character, a character-string, an arithmetic or logical expression or a built-in function.

4) If **x** is a character-string, it must not be greater than 4 bytes in length.

5) **y** should have a value of 0 through 32.

6) If **y** is 32, the result will be all zeroes regardless of the value of **x**.

7) If parameters are of *DEC or *NCS data types, fractional digits will be truncated when the conversion to **BIT**-string takes place.

8) All data types are treated as unsigned **BIT**-string values.

```
:   DCL daw BIN,          :
:       val BIN;          :
:   daw = 6;              :
:   val = SHL (daw,3);    :
:                         :
```

In the above, **val** becomes a **BIT**-string **00110000**, (value 48.)

SHR Built-in Function

```
SHR (x, y)
```

The **x** parameter is converted to a 32 bit **BIT**-string, if necessary, and is logically shifted **y** bits to the right. Zeroes are introduced in the vacated logically bit positions on the left.

General Rules for the SHR Built-in Function:

1) **x** and **y** are required parameters which must be arithmetic, logical or relational expressions.

2) **x** and **y** may be subscripted and/or pointer qualified.

3) If, when converted to **BIT**-string, **x** is less than 32 bits in length, zero bits are added on the left. **x** may be a character, a character-string, an arithmetic or logical expression or a built-in function.

4) If **x** is a character-string, it must not be greater than 4 bytes in length.

5) **y** should have a value of 0 through 32.

6) If **y** is 32, the result will be all zeroes regardless of the value of **x**.

7) If parameters are of *DEC or *NCS data types, fractional digits will be truncated when the conversion to **BIT**-string takes place.

8) All data types are treated as unsigned **BIT**-string values.

```
:   DCL doe BIN,          :
:       val BIN;          :
:   doe = 48;             :
:   val = SHR (doe,3);    :
:                         :
```

In the above, **val** becomes a **BIT**-string **00000110**, (value 6.)

BSTR (pseudo-variable) Built-in Function

The **BSTR**, **CSTR** and **NSTR** functions can be used to request explicitly that a field is used in a manner that may differ from the data declaration.

```
BSTR   (a   (,o1   (,o2)   )   )
            (      (   )   )
            (              )
```

The use of **BSTR** causes a to be treated as a **BIT**-string, regardless of how it was declared. **o1** specifies the first bit position, from the left, to be considered. If not included, it defaults to one (1). **o2** specifies the length of the field, in bits. If not included, it is assumed to define the rest of field a (from the first bit considered to the rightmost bit, inclusive, in the field).

General Rules for the **BSTR** Built-in Function:

1) **a** is a required parameter which must be a constant, a variable or a structure.

2) **a** cannot be subscripted but may be pointer qualified.

3) **o1** and **o2** are optional parameters that must be **BINARY** literals.

4) If **o1** is greater than the declared size of **a**, then **o2** must be included.

5) If **o1** is included and **o2** is omitted, then **o1** must be less than or equal to the declared (default) size of **a**.

6) If **o1** is omitted, **o2** must also be omitted.

7) **o2**, if included, must have a value of 1 through 32.

```
:   DCL cub CHAR(2),          :
:       num BIT(4),           :
:       val BIT(2);           :
:   cub = 'sa';               :
:   num = BSTR (cub,4,4);     :
:   val = BSTR (num,2,2);     :
```

```
:                                   :
```

The above statement: **num = BSTR (cub,4,4);** would result in **num** becoming a **BIT**-string **0001**.

The above statement: **val = BSTR (num,2,2);** would result in **val** becoming a **BIT**-string **00**.

In the above example, the use of **BSTR** to isolate bits of a field is demonstrated. The following example demonstrates the use of **BSTR** to treat a field as if it had the **BIT** attribute.

```
:   BSTR (EBW000,1,8) = '80'x;   :
:                                :
```

In this example the declared data-type of EBW000, which is **CHAR(1)**, is being temporarily overridden. Since a hex 80 is to be moved into EBW000, EBW000 must be given a data-type compatible with the assignment.

CSTR (pseudo-variable) Built-in Function

```
CSTR   (a   (,e   (,o )   )   )
            (,v1  (,v2)   )
```

The use of **CSTR** causes a to be treated as a character-string, regardless of how it was declared. **e** or **v1** specifies the first byte position from the left to be considered. If not included, it defaults to one (1), (the first byte of **a**). **o** or **v2** specifies the length of the field, in bytes. If not included, it is assumed to define the rest of field **a** (from the first byte considered, to the rightmost byte, inclusive, in the field).

General Rules for the **CSTR** Built-in Function:

1) **a** is a required parameter which must be a constant, a variable or a structure.

2) **a** cannot be subscripted but may be pointer qualified.

3) **e** must be a literal if **o** is not included. If **o** is included, **e** may not be subscripted or may not be a pseudo-variable. It may be a literal, a constant, a variable, a structure, an expression or a function call.

4) **o** is an optional parameter that must be a **BINARY** literal.

5) If **e** is specified and **o** is omitted, **e** must be less than or equal to the declared (default) size of **a**. (If **o** is not omitted, there is no restriction on the size of **e**.)

6) If **e** is omitted, **o** must also be omitted.

7) **o**, if included, must have a value of 1 through 4087.

```
:   DCL suffix BIT(16),              :
:       val CHAR(2);                 :
:   suffix = '0110010111100010'B;    :
:   val = CSTR (suffix,2);           :
:   CSTR(val,2) = CSTR (val,1,1);    :
:                                    :
```

The above statement: **val = CSTR (suffix,2);** would result in **val** becoming the character-string '**S** ' ('**S**' and a blank).

The above statement: **CSTR(val,2) = CSTR(val,1,1);** would result in **val** becoming the character-string '**SS**'.

8) **v1** is a variable starting point that must be declared as **BINARY**.

9) **v2** is a variable length that must be declared as **BINARY**.

10) **v1** and **v2** can only be used in assignment statements.

11) **v2** can only be used on the right side of assignment statements.

12) **EXCEPTION**: Within an **IF** statement **V1** and **V2** may be used only on the left side of the equal sign. Nesting with other built-in functions is not allowed.

EXAMPLES:

```
:   INIT          = CSTR(NAME,STPNT,VARLTH);     :
:   CSTR(name,10) = CSTR(INIT,STPNT,VARLTH);     :
:   CSTR(name,10) = CSTR(INIT,4,VARLTH);         :
```

NSTR (pseudo-variable) Built-in Function

```
NSTR   (a   (,e   (,o)   )   )
            (     (  )   )
            (            )
```

The use of **NSTR** causes a to be treated as a Numeric character-string with an implied decimal point to the right of the field, regardless of how it was declared. The function disregards the sign. **e** specifies the first byte position, from the left, to be considered: if not included, it defaults to one (1). **o** specifies the length of the field, in bytes. If **o** is not included, a default value is assumed that defines the rest of field (from the first byte considered, to the rightmost byte, inclusive, in the field).

General Rules for the **NSTR** Built-in Function:

1) **a** is a required parameter which must be a constant, a variable or a structure.

2) **a** cannot be subscripted but may be pointer qualified.

3) **e** is an optional parameter and represents the most general of operands which can be a literal, a constant, a variable, a structure, an array element, an expression or a function call. **EXCEPTION**: if **o** is omitted, **e** must be a literal. The format of the expression **e** must conform to the rules governing subscripts (See Chapter 4, Subscripts), with the exception that variables may not be pseudo-variables.

4) **o** is an optional parameter that must be a **BINARY** literal.

5) If **e** is specified and **o** is omitted, **e** must be less than or equal to the declared (default) size of **a**. (If **o** is not omitted, there is no restriction on the size of **e.**)

6) If **e** is omitted, **o** must also be omitted.

7) **o**, if included, must have a value of 1 through 256.

Example:

```
:   DCL  cur CHAR (6),      :
:        val PIC  '999';    :
:   cur = '123456';         :
:   val = NSTR(cur,3,3);    :
:                           :
```

As a result, **val** would become 345. The internal representation of **val** (in hexadecimal notation) would be **F3F4F5.**

```
:   DCL amt CHAR(5);        :
:   DCL decamt DEC(5,2);    :
:   decamt = NSTR (amt);    :
```

In the above example, unless **amt** contains data in zoned **DECIMAL** format, the statement

```
decamt = NSTR (amt);
```

will not be meaningful at execution time. The execution of the statement will be as in *DEC = *NCS assignment statements, i.e., the data in **amt** will be packed first, then the decimal points will be aligned, and the data moved. If **amt** contained the hex value **F0F0F2F4F1**, then the result of the assignment would be that **decamt** contains **00241F**. The following is probably not meaningful:

```
:   DCL bitamt BIT(16);     :
:   DCL decamt DEC(5,2);    :
:   bitamt = '4e60'x;       :
:   decamt = NSTR (amt);    :
```

VSTR Built-in Function

```
VSTR   (a, e1, e2)
```

The **VSTR** function differs from the pseudo-variables **BSTR**, **CSTR** and **NSTR** in that the size of the field to be considered may be variable. In other words, parameter **e2** need not be a literal. Additionally, all the parameters of the **VSTR** function are required. There are no defaults, as in the pseudo-variable functions.

A **VSTR** function specification must appear, by itself, as the sending field (right hand side) of an Assignment statement.

The field **a** will be treated as the data type it was originally declared when the rules of assignment are applied. However, all calculations are performed on a byte basis, and the assignment is performed as a straight byte for byte move without data conversions of any kind. **e1** specifies the first position, from the left, to be considered. **e2** specifies the length of the field, in bytes.

The **VSTR** function should only be used when the size of the move is variable and no data conversion is required. Use of the pseudo-variable functions results in the generation of more efficient BAL code when the size of the move is not variable.

```
| element variable |  =   VSTR function  ;
| pseudo-variable  |
| structure        |
| array element    |
-                  -
```

```
:    DCL msg CHAR(150),        :
:        pge BIN,              :
:        val CHAR(200);        :
:    pge = 125;                :
:    val = VSTR(msg,1,pge);    :
```

In the above example, the leftmost 125 characters of **msg** are moved into the leftmost 125 bytes of **val**. The rest of **val** is not affected.

General Rules for the **VSTR** Built-in Function:

1) **a** is a required parameter that may be any data type: a literal, a constant, a variable, a structure, an expression or a function call.

2) **a** cannot be subscripted, but may be pointer qualified.

3) **e1** is a required parameter that must be an expression and may be subscripted. It should yield a result whose value is 0 through 256. The expression must conform to the rules governing subscripts (See Chapter 4, Subscripts), with the exception that a variable may not be a pseudo-variable.

4) **e2** is a required parameter and must be a literal, a variable that may be subscripted or an expression.

5) **e2** should yield a value of 0 through 256.

6) If the value of **e2** is negative, the result is unpredictable.

7) If the value of **e2** is zero, no data will be moved.

8) Overflow will result if the receiving field is shorter than the length of the move specified by **e2**.

9) The receiving field will not be filled with blanks if it is larger than the sending (**VSTR**) field.

```
:   DCL header CHAR(10),              :
:       length BIN,                   :
:       slot2 CHAR(10),               :
:       slot3 CHAR(10);               :
:   slot2 = VSTR(header,1,length);    :
:                                     :
```

In the above example, if the value of **length** is greater than 10, then field **slot3** will be overlaid during execution. There are two ways to avoid this:

A) An additional field may be declared following slot2 so that any overflow will not overlay slot3.

B) Rewrite the assignment as follows:

```
:   slot2 = VSTR(header,1,MIN(length,10));    :
:                                             :
```

This way, if **length** is greater than 10 it will not be used and field **slot3** will not be overlaid.

Other tests may be necessary such as preventing a zero or negative value from being used in **VSTR** as the **e2** parameter. If such a number is used, it could result in a very long move that would overlay areas following the receiving field.

ADDR Built-in Function

```
ADDR (a)
```

This function returns a pointer to the data item **a**. (Refer to Chapter 4 for a description of pointers).

General Rules for the **ADDR** Built-in Function:

1) a must be an identifier, either an element variable or a constant. It may be subscripted and/or pointer qualified.

2) a may not be an expression.

```
:   %INCLUDEAF eboeb;              :
:   DCL 1 input BASED (inptr),     :
:         2 task CHAR(30),         :
:         2 charter CHAR(30),      :
:         2 flds (20) BIN(15);     :
:   DCL aptr POINTER;              :
:   GETCC d2,l1;                   :
:   inptr = ce1cr1;                :
:   aptr = ADDR (flds(1));         :
:                                  :
```

The above will result in the address of **flds** being stored into the **POINTER** called **aptr**.

Notes:

1) If the address of a field is to be obtained using relative addressing, the following method is satisfactory:

```
:   fldptr = ADDR (flds) + displacement;    :
:                                           :
```

2) This function may not be used to obtain the address of a statement label, as in:

```
:   mylab: sal = wge + bon;   :
:          ...                :
:   val = ADDR (mylab);       :
:          ...                :
```

3) It (and all other non-pseudo variable built-in functions) also cannot be used for register value storing in a TPF macro statement or a user macro statement. The following is not allowed:

```
:   ENTRC Z009(#RG1=ADDR(fld));   :
:                                 :
```

An alternative coding method is as follows:

```
:   otherfld = ADDR(fld);        :
:   ENTRC Z009(#RG1=otherfld);   :
:                                :
```

CASE Built-in Function

```
CASE (x,y)
```

The above causes a branch based on the value of **x**.

General Rules for the **CASE** Built-in Function:

1) **CASE** may only be used in an **IF - THEN DO** statement.

2) **x** must be a **BINARY** variable.

3) **y** must be an **BINARY** literal.

4) The **IF - THEN DO** statement must be followed by at least a number of either **GOTO** statements; or **CALL** and/or ENTER statements indicated by parameter y.

5) If **x** is less then 1 or **x** is greater than parameter **y**, a branch to the last **GOTO**, **CALL** or ENTER routine will occur.

6) **GOTO** statements and **CALL** statements may NOT be mixed in a single **CASE** expression.

7) **GOTO** statements and ENTER statements may NOT be mixed in a single **CASE** expression.

8) **CALL** and ENTER may be freely mixed within a single **CASE** expression.

9) ENTRC is ONLY valid as the last statement in **CASE** as indicated by **y**. ENTNC and ENTDC may be used anywhere within **CASE**.

10) Parameters may NOT be passed in the called procedures or in ENTER(s) within a **CASE** expression. **CALL PROCEDURE1(B);** or **ENTNC (#rg1=var);** will generate an error message.

```
:   DCL (CHOICE)   BIN(31);          :
:   CHOICE = 3;                      :
:   IF CASE(CHOICE,5) THEN DO;       :
:       GOTO ROUTINE1;               :
:       GOTO ROUTINE2;               :
:       GOTO ROUTINE3;               :
:       GOTO ROUTINE4;               :
:       GOTO ROUTINE5;               :
:    END;                            :
:                                    :
```

The above will cause a branch to **ROUTINE3**.

```
:   DCL (CHOICE)        BIN(31);     :
:   CHOICE = 3;                      :
:   IF CASE(CHOICE,5) THEN DO;       :
:       CALL ROUTINE1;               :
:       CALL ROUTINE2;               :
:       CALL ROUTINE3;               :
:       CALL ROUTINE4;               :
:       CALL ROUTINE5;               :
:    END;                            :
:                                    :
```

The above will cause a branch to the **CALL** to **ROUTINE3** and on returning from **ROUTINE3** will branch to the **END** of the **CASE** statement.

```
:   DCL (CHOICE)        BIN(31);      :
:   CHOICE = 3;                       :
:   IF CASE(CHOICE,5) THEN DO;        :
:       ENTDC  PROG1;                 :
:       CALL   ROUTINE2;              :
:       ENTNC  PROG3;                 :
:       CALL   ROUTINE4;              :
:       ENTRC  PROG5;                 :
:    END;                             :
:                                     :
```

The above will cause a branch to the ENTNC to **PROG3**. Since this is a no return enter, no branch to the **END** of the **CASE** statement is necessary.

Examples:

```
         DECLARE  (A,B)   BIN(31);
         DECLARE  CHOICE  BIN(31);
 1)  IF CASE(A,6) THEN DO;
                    L     R15,A$(R7)
                    LTR   R15,R15
                    BNH   *+12
                    CL    R15,=XL004'00000006'
                    BNH   *+8
                    LA    R15,0006
                    BCTR  R15,R0
                    SLL   R15,2
                    B     *+4(R15)
        GOTO ROUTINE1;
                    B     ROUTINE1
        GOTO ROUTINE2;
                    B     ROUTINE2
        GOTO ROUTINE3;
                    B     ROUTINE3
        GOTO ROUTINE4;
                    B     ROUTINE4
        GOTO ROUTINE5;
                    B     ROUTINE5
        GOTO ERROR_RT;
                    B     ERROR$RT
     END;

 2)   IF CASE(A,5) THEN DO;
                    L     R15,A$(R7)
                    LTR   R15,R15
                    BNH   *+12
                    CL    R15,=XL004'00000005'
                    BNH   *+8
                    LA    R15,0005
                    BCTR  R15,R0
                    SLL   R15,3
                    B     *+4(R15)
        CALL ROUTINE1;
                    BAS   R15,ROUTINE1
                    B     $GEN0001
        CALL ROUTINE2;
                    BAS   R15,ROUTINE2
                    B     $GEN0001
        CALL ROUTINE3;
```

```
                    BAS   R15,ROUTINE3
                    B     $GEN0001
        CALL ROUTINE4;
                    BAS   R15,ROUTINE4
                    B     $GEN0001
        CALL ROUTINE5;
                    BAS   R15,ROUTINE5
   END;
          $GEN0001  DS    0H
```

```
3)   IF CASE(CHOICE,5) THEN DO;
                    L     R15,CHOICE$(R7)
                    LTR   R15,R15
                    BNH   *+12
                    CL    R15,=XL004'00000005'
                    BNH   *+8
                    LA    R15,0005
                    BCTR  R15,R0
                    SLL   R15,3
                    B     *+4(R15)
        ENTDC  PROG1;
                    ENTDC PROG1
        CALL   ROUTINE2;
                    BAS   R15,ROUTINE2
                    B     $GEN0001
        ENTNC  PROG3;
                    ENTNC PROG3
        CALL   ROUTINE4;
                    BAS   R15,ROUTINE4
                    B     $GEN0001
        ENTRC  PROG5;
                    ENTRC PROG5
     END;
                    $GEN0001  DS    0H

        EXITC;
                    EXITC

      ROUTINE2: PROC;
         ROUTINE2  DS    0H
                   ST    R15,$SAV0001(R7)
         ENTRC PROG2 (#R1=A,B=#R2);
                   L     R1,A$(R7)
                   ENTRC PROG2
                   ST    R2,B$(R7)
         RETURN;
                   L     R14,$SAV0001(R7)
                   BR    R14
      END ROUTINE2;
```

The table in figure 5.1 depicts the relationships between the (previously discussed) built-in functions if nesting them is under consideration. Nesting, here, refers to the use of a function specification in its entirety, as a parameter of another function. Outer functions are read by row, inner functions are read by column.

LSTR Built-in Function

```
a  =  LSTR     (b  (,c))
```

This function returns the length of a data field.

General Rules for the **LSTR** Built-in Function:

1) **a** is a receiving field which must be declared as either **BINARY** or **BIT**.

2) **b** is a data field, a required parameter that must be a variable or an array. **b** may NOT be declared as **BIT**.

3) **c** is an optional parameter which can be a variable or a constant, and is used as follows:

- Starting point, which may be either a variable or a constant, if **b** is a structure. If the variable is an identifier in the structure of the data field then it represents the displacement from where the calculation of the length is made. If the variable is not in the structure of the data field then its content will used to calculate the length.

- Index, which must be a constant. If **b** is an array and:
  - at level 01 then the default is the size of the entire array.
  - at a sub-structure then the default is the size of an entry of the array.

- If b is a field of a sub-structure in an array then the optional parameter must be omitted.

Example:

```
:         DECLARE (J,K)        BIN;                      :
:         DECLARE 01 GAME (4),                           :
:                 02 NAME      CHARACTER (10),           :
:                 02 ADDRESS   CHARACTER (10);           :
:         DECLARE 01 ONPUT,                              :
:                 02 ON1,                                :
:                 03 ON2 (3),                            :
:                 04 ONE       CHARACTER (4),            :
:                 04 TWO       CHARACTER (8),            :
:                 03 ON3       CHARACTER (9);            :
:         DECLARE 01 KAME,                               :
:                 02 KAM       CHARACTER ( 4),           :
:                 02 LAM       CHARACTER ( 6),           :
:                 02 ADDR      CHARACTER (10);           :
:  /*  data field is a structure                  */     :
:        K     = LSTR (KAME);                            :
:                LA    R15,0020                          :
:                STH   R15,K$(R7)                        :
:  /*  data field is a substructure               */     :
:        K     = LSTR (ON1);                             :
:                LA    R15,0045                          :
:                STH   R15,K$(R7)                        :
:  /*  and c is a constant                        */     :
:        K     = LSTR (KAME,2);                          :
:                LA    R15,0018                          :
:                STH   R15,K$(R7)                        :
:  /*  variable c is in a structure of data field    */   :
:        K     = LSTR (KAME,LAM);                        :
:                LA    R15,0016                          :
:                STH   R15,K$(R7)                        :
:  /*  variable c is not in a structure of data field*/   :
```

```
:         K     = LSTR (KAME,J);                          :
:                 LA    R15,0020                          :
:                 LH    R14,$J(R7)                        :
:                 CR    R15,R14                           :
:                 BL    $GEN0001                          :
:                 SR    R15,R14                           :
:                 STH   R15,$K(R7)                        :
:                 $GEN0001   DS  0H                       :
:  /*  Default is a size of an entire array        */     :
:  /*  because GAME is a level 01                  */     :
:         K     = LSTR (GAME);                            :
:                 LA    R15,0080                          :
:                 STH   R15,$K(R7)                        :
:  /*  level is not 01 then the default is the     */     :
:  /*  length of an entry                          */     :
:         K     = LSTR (ON2);                             :
:                 LA    R15,0012                          :
:                 STH   R15,$K(R7)                        :
:  /*  length of an entry in an array.             */     :
:         K     = LSTR (GAME,1);                          :
:                 LA    R15,0020                          :
:                 STH   R15,$K(R7)                        :
:  /*  length of an entry in an array, same as above*/    :
:  /*  a field of a sub-structure in an array       */    :
:         K     = LSTR (NAME)                             :
:                 LA    R15,0010                          :
:                 STH   R15,$K(R7)                        :
:         K     = LSTR (ONE);                             :
:                 LA    R15,0004                          :
:                 STH   R15,$K(R7)                        :
```

BSTM Built-in Function

```
a   =  BSTM     (b , c)
```

This function returns a condition code.

General Rules for the **BSTM** Built-in Function:

1) **a** is the result field, which must be declared as **BIN** or **BIT**, and will receive a condition code setting as follows:

   0 if all the tested bits are off (or zero).
   1 if some of the tested bits are on, some off (or mixed).
   3 if all the tested bits are on (or one).

2) **b**, the tested identifier, is a required parameter that must be a variable. If the length of the tested identifier is more than a byte, then the first byte is compared as the default. The Programmer may use the **ADDR** built-in function to allocate a particular byte as shown in example 3 below.

3) **c**, the immediate byte, is used as a mask for selecting the bits to be tested.

Example:

```
             DECLARE B            DEC(7,2),
                     C            CHAR(4),
                     J            BIN(31);

 1)   FLAG = BSTM((ADDR(B)+2),'10010111'B);
                      LA    R15,B$(R7)
                      A     R15,=XL004'00000002'
                      SR    R14,R14
                      TM    0(R15),X'97'
                      BZ    $GEN0011
                      LA    R14,1
                      BM    $GEN0011
                      LA    R14,3
            $GEN00011 DS    0H
                      STH   R14,FLAG$(R7)


 2)    FLAG = BSTM(C,'100101'B);
                      SR    R15,R15
                      TM    C$(R7),X'25'
                      BZ    $GEN0007
                      LA    R15,1
                      BM    $GEN0007
                      LA    R15,3
            $GEN0007  DS    0H
                      STH   R14,FLAG$(R7)

3)  FLAG = BSTM((ADDR(C)+3),'00001101'B);
                      LA    R15,C$(R7)
                      A     R15,=XL004'00000003'
                      SR    R14,R14
                      TM    0(R15),X'0D'
                      BZ    $GEN0012
                      LA    R14,1
                      BM    $GEN0012
                      LA    R14,3
            $GEN0012  DS    0H
                      STH   R14,FLAG$(R7)

 4)  IF BSTM(J,'100101'B) = 1 THEN GO TO NEXT0001;
                      SR    R15,R15
                      TM    J$(R7),X'25'
                      BZ    $GEN0013
                      LA    R15,1
                      BM    $GEN0013
                      LA    R15,3
                     $GEN0013  DS    0H
                      C     R15,=XL004'00000001'
                      BE    NEXT0001
```

NOTE: If the **BSTM** is used in an assignment statement, the result field may be tested in a separate **IF** statement. If the **BSTM** is used in an **IF** statement, and the result will be used in a test evaluation, see example 4.

. . o . i u . n t . n e . e r . r	ABS	MAX	MIN	MOD	SIGN	ALPHA	NUMERIC	INDEX	SHL	SHR	BSTR	CSTR	NSTR	VSTR	ADDR	LSTR	BSTM	CASE	ROUND
ABS	1	1	1	1	1	1	1	1	1	1	1	-	1	-	1	-	-	1	
MAX	1	1	1	1	1	1	1	1	1	1	1	-	1	-	1	-	-	1	
MIN	1	1	1	1	1	1	1	1	1	1	1	-	1	-	1	-	-	1	
MOD	1	1	1	1	1	1	1	1	1	1	1	-	1	-	1	-	-	1	
SIGN	1	1	1	1	1	1	1	1	1	1	-	-	1	-	1	-	-	1	
ALPHA	-	-	-	-	-	-	-	-	-	-	1	1	1	-	-	-	-	-	
NUMERIC	-	-	-	-	-	-	-	-	-	-	1	1	1	-	-	-	-	-	
INDEX	-	-	-	-	-	-	-	-	-	-	2 3	2 3	2 3	-	-	-	-	-	
SHL	1	1	1	1	1	1	1	1	1	1	1	2	1	-	1	-	-	-	
SHR	1	1	1	1	1	1	1	1	1	1	1	2	1	-	1	-	-	-	
BSTR	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
CSTR	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
NSTR	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
VSTR	4	4	4	4	4	4	4	4	4	4	4	-	4	-	4	-	-	-	
ADDR	-	-	-	-	-	-	-	-	-	-	1	1	1	-	-	-	-	-	
LSTR	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
BSTM	-	-	-	-	-	-	-	-	-	-	-	-	-	-	2	-	-	-	-
CASE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ROUND	1	1	1	1	-	-	-	-	-	-	-	-	-	-	-	-	-	1	

Key:
Dash indicates illegal combination of data types.
1 - Permitted.
(As any parameter, if function accepts more than one.)
2 - Permitted as first parameter.
3 - Permitted as second parameter.
4 - Permitted as third parameter.

Figure 5.1 Nesting of Built-in Functions.

PROGRAMMER-DECLARED FUNCTIONS

A programmer may specify that an identifier denote a programmer-declared function by coding the identifier in a **DECLARE** statement with the attribute **FUNCTION**.

General Format:

```
 _          _
| DCL      |  identifier   FUNCTION   ;
| DECLARE  |
-          -

:   DCL  sqroot  FUNCTION;    :
:                             :
```

The programmer may then code, elsewhere in his program, a **PROCEDURE** statement using this identifier as the **PROCEDURE** label:

```
:   sqroot: PROC (a);    :
:                        :
```

The **PROCEDURE** statement contains a list for all parameters that may be passed to the programmer-declared function by the invocation.

The programmer terminates the declared function logically with a parameterized **RETURN** statement and then physically with an **END** statement.

An example of the use of a programmer-declared function:

```
:   DCL      sqroot FUNCTION;          :
:            ...                       :
:            result = sqroot(item);    :
:            ...                       :
:   sqroot:  PROC(invalue);            :
:            ...                       :
:            RETURN(outvalue);         :
:            END;                      :
:            ...                       :
```

In this example the function **sqroot** is invoked to compute the square root of **item**.

General Rules for Programmer-declared Functions

1) There is no restriction on the number of parameters that may be passed to a programmer-declared function. However, at least one parameter must be specified when the function is invoked.

2) A parameter may be one of the following:

   - variable
   - structure

3) A parameter may not be subscripted nor pointer qualified.

4) Normal exit is through one **RETURN** statement with one mandatory parameter, enclosed in parentheses, coded after it. It is possible to exit by use of a **GOTO** statement, but the value computed in the function will be lost, and the execution of the statement that contained the corresponding function invocation reference is discontinued.

5) Programmer-declared functions may be invoked by stand-alone sending fields in assignment statements, or by operands in expressions of any type.

6) Programmer-declared functions are invoked by coding the function name with the parameters to be passed, enclosed in parentheses, immediately following.

7) A maximum of 32 programmer-declared functions are permitted in a program.

8) A programmer-declared function may refer to all data names in the program without giving separate data declarations.

9) Entry to a programmer-declared function does not cause allocation of another block of **AUTOMATIC** storage.

10) The (mandatory) value returned is in a register. If the **RETURN** statement parameter is an arithmetic data type other than **BINARY**, a conversion to **BINARY** will take place and data may be lost.

11) The **END** statement terminating the function may not have an operand following it.

PROCEDURES

A **PROCEDURE** is a group of statements headed by a **PROCEDURE** statement and terminated by an end statement. There are two types of procedures: Internal and External. (Programmer-declared functions are headed by a **PROCEDURE** statement and terminated by an **END** statement, however, programmer-declared functions are classified as functions rather than procedures).

General Format of PROCEDURE statements:

For Internal Procedures:

```
label:  | PROC      |  ( ( parameter list ) )  ;
        | PROCEDURE |  (                     )
        -           -  (                     )
```

For External Procedures:

```
label:  | PROC      |  ;
        | PROCEDURE |
        -           -
```

NOTE: Each parameter may be an unsubscripted variable or a structure name, but may not be a pseudo-variable.

Internal Procedures

Optional internal procedures may be coded within the external procedure. Each one must have a **PROCEDURE** and **END** statement, just as external procedures. A list of parameters may be coded in the **PROCEDURE** statement heading the internal **PROCEDURE**.

General Format of the **PROC** statement:

```
intrnl_proc_lbl:  | PROC      |  (  (arg1 (,arg2,...argn)  )  )  ;
                  | PROCEDURE |  (                           )
                  -           -  (                           )
```

The logical end of an internal **PROCEDURE** is denoted by a **RETURN** statement.

General Format of the **RETURN** statement:

```
RETURN  ;
```

The physical end is denoted by an **END** statement.

General Format of the **END** statement:

```
END  ( internal_proc_label )  ;
     (                     )
```

Example of an internal procedure:

```
intproc1:  PROC(pay,rate,date) ;
           ...
           RETURN ;
           END ;
```

Upon execution of a **RETURN** statement, control will be transferred to the statement following the **CALL** statement that invoked the internal **PROCEDURE**.

An internal **PROCEDURE** is invoked by coding the label prefixing its **PROCEDURE** statement in a **CALL** statement. If a list of corresponding parameters is to be passed, it must be enclosed in parentheses immediately following the coded label. The **CALL** and **PROCEDURE** parameters must have similar or identical data types, that is:

CALL PARAMETER	PROCEDURE PARAMETER
CHAR	CHAR
BIN/DEC/BIT	BIN/DEC/BIT
LAB	LAB

Format of the **CALL** statement:

```
CALL  internal_proc_label  (  (arg1 (,arg2,...argm)  )  )  ;
                           (                           )
```

Example of a **CALL** statement:

```
CALL intproc1( pay2,rate,date2) ;
```

General Rules for Internal Procedures:

1) An internal **PROCEDURE** may invoke another internal **PROCEDURE**.

2) An internal **PROCEDURE** may not be recursive, i.e., should not invoke itself, directly or indirectly.

3) The compiler does not generate a branch around an internal **PROCEDURE**; it has to be placed outside the logic flow of the main program.

4) There may be more than one **RETURN** statement per internal **PROCEDURE**.

5) Exit from an internal **PROCEDURE** should be through the use of a **RETURN** statement.

6) The **RETURN** statement logically ending an internal **PROCEDURE** may not have a parameter coded with it.

7) An internal **PROCEDURE** can only be referenced through a **CALL** statement.

8) Entry to an internal **PROCEDURE** does not cause allocation of another **AUTOMATIC** storage block.

9) An internal **PROCEDURE** may refer to all data names in the program without giving separate data declarations.

10) A maximum of 50 internal procedures may be coded in a program.

11) An internal **PROCEDURE** may not be physically nested within another in the source program, although an internal **PROCEDURE** may contain **CALL'**s to other internal procedures in the program.

12) Each internal **PROCEDURE** must have a corresponding **END** statement.

13) Parameters in the internal **PROCEDURE** should be short to conserve storage and increase program efficiency.

14) The parameter for a **CALL** of an internal **PROCEDURE** statement can be used to develop values, unlike the parameter of the **PROCEDURE** statement that merely specifies a field into which the value is placed.

The parameters of the **PROCEDURE** are limited to structured variables that may not be signed, subscripted or pointer-qualified. The parameters of the **PROCEDURE CALL** have a less limited format and may be

- structured, arrayed, signed, subscripted, concatenated, or pointer-qualified
- variables, constants, literals, built-in functions, pseudo-variables, or programmer-declared functions
- arithmetic, relational, or logical expressions

They may NOT be system-equate names.

It is a good idea to use pointers when passing large fields, rather than the fields themselves. This can be illustrated by the following:

```
:   /*  poor method  */      :
:   DCL xstring CHAR(150);    :
:   DCL ystring CHAR(150);    :
:   CALL subr (xstring);      :
:        ...                  :
:   subr: PROC (ystring);     :
:        ...                  :
:   RETURN;                   :
:   END subr;                 :
:                             :
```

This requires 150 characters for **ystring** and will result in a long move for each execution of the **CALL**. An alternate method would be:

```
:   /*  better method  */              :
:   DCL xstring CHAR(150);              :
:   DCL ystring CHAR(150) BASED (p);    :
:   CALL subr (ADDR(xstring));          :
:        ...                            :
:   subr: PROC (p);                     :
:        ...                            :
:   RETURN;                             :
:   END subr;                           :
:                                       :
```

This results in the address of **xstring** being stored in **p**. **subr** may now reference **ystring**. It will, in fact, be referencing **xstring** through the pointer **p**.

External Procedures

The **PROCEDURE** statement must be the first physical statement in the program. The entire program constitutes an external procedure. The program name serves as the label on the **PROCEDURE** statement for the external procedure. This label should conform to local programming standards for program names. External procedures are activated in the same manner as any other program segment in TPF, that is, through the use of an enter-type macro.

The external **PROCEDURE** is physically terminated by an **END** statement (See End Statement, this chapter) Parameters can be passed to the external **PROCEDURE** by the use of register-loading statements coded in conjunction with the invoking ENTER-type macro. The **START** statement must be used in the external **PROCEDURE** receiving the parameters to share the values passed in the registers. The optional parameter list following the **PROCEDURE** statement, as illustrated at the beginning of this section, is to be used only with internal procedures. If parameters are being passed via registers to the external **PROCEDURE**, the **START** statement (See **START** Statement, this chapter) should be the first executable statement in the program, following the **PROCEDURE** statement. **DECLARE** statements are non-executable and may, therefore, precede the **START** statement, if desired. External procedures may invoke other external procedures or Assembler Language programs in the system by the use of an ENTER-type macro. There is no restriction placed upon activating an Assembler Language program from a SABRETALK program or vice versa.

MACROS

Macros (names of macro requests) can be classified as:

A) TPF macros (supplied by IBM).
B) SABRETALK Macros (**START**, **GLOBX**, etc.).
C) Application macros (CSERA, TDLYC, etc.).
D) Special macros (TPFDF, ETC.)
E) User macros.

The TPF system contains a set of macros that, at execution, handle work such as:

- Main storage management.
- Input / output functions.
- Queuing of work to be done.

Most of these macros are coded in the same way under SABRETALK and under TPF. However, macros which presume that parameters are set up in registers, or will return control with information in registers, must be coded as part of a macro statement that will indicate the corresponding identifiers in the SABRETALK program. Parameters may be passed in any register allocated for applications under TPF. They may be referred to by number (0, 1, 2, R0, R1, R2, etc) or by the TPF system equates (RAC, RG1, RGA, etc.). Parameters to be passed to registers are coded first, followed by parameters received from registers.

It should be noted that the contents of the macro table and the status of every macro, is determined by each installation. The table is established by the user when the Compiler system is installed. Thereafter, it is maintained via the SABRETALK utility UPDATMAC. Programmers should be provided with a list of macros that are supported by the Compiler at their work site.

General Format of Macro Statements:

To Load Registers:

```
macro (arg1,arg2,...,arg20) (optional register loading, )  ;
      (                   ) (storing: see Section 12.5.1)
      (                   ) (                           )

  (where arg = macro argument)
```

Each argument will be an identifier. One argument may specify a statement label to be used as an error-out address.

```
:    ENTRC program_name(#R0=parm_1,#R3=parm_2,     :
:                       #R4=parm_3,                :
:                       parm_4=#R2,parm_5=#5) ;    :
:                                                  :
```

This example shows how a macro statement is used to enter a TPF program segment that expects parameters in #R0, #R3 and #R4, and that returns parameters in #R2 and #R5. Although SABRETALK provides the facility, parameters need not be passed in registers; they may also be passed in the ECB or in blocks attached to the ECB.

General Rules for Macro Statements

1) Any macro statement that passes parameters through the ECB (Entry Control Block) can have these parameters set up by the program before the execution of the macro statement. (The **ADDR** built-in function is unacceptable as a macro parameter).

   An example:

```
:   level = ADDR(ce1fa2);      :
:   ENTRC FACE(#R0=ordno,      :
:              #R6=#ss1ri,     :
:              #R7=level,      :
:              rtncod=#R0);    :
:                              :
```

2) All parameters passed via register loading on the macro statement are coded first, followed by all parameters being stored on the macro statement.

3) A maximum of twenty (20) macro arguments is allowed.

4) There is no limit to the number of parameters passed (or loaded into registers) and/or returned (or stored from registers). Non-pseudo variable built-in functions cannot be used for obtaining register values within a TPF macro statement or a user macro statement. The following is not allowed:

```
:   ENTRC Z009(#R1=ADDR(fld));    :
:                                 :
```

An alternative coding method is as follows:

```
:   otherfld = ADDR(fld);         :
:   ENTRC Z009(#R1=otherfld);     :
:                                 :
```

5) SABRETALK does not support the GLOBZ macro. When referencing global fields it is necessary to code the **GLOBX** macro or **GLOBW** macro as shown in the following macro statement:

```
:   DCL @fnc00 BIN(31) BASED(fonptr);  :
:   GLOBX @fnc00(fonptr=#R15);         :
:                                      :
```

This macro statement will conditionally generate a set of Assembler Language instructions (including the GLOBZ macro) in its expansion. The following is the expansion of the **GLOBX** statement:

```
:   GLOBZ REGR=#R15(FLD=@fnc00)  :
:                 (          )   :
:   LA    R15,@fnc00             :
:   DROP  R15                    :
:   ST    R15,fonptr$(R7)        :
:                                :
```

As shown in the above examples, the register specified in the **GLOBX** statement must be RDB (R15). The identifier (in this case, called **fonptr**) must be defined by the programmer in a **DECLARE** statement as shown by the following:

```
:  DCL 1 fonitms(8) BASED(fonptr),   :
:      2 foncor POINTER,             :
:      2 fonfil BIT(32);             :
:                                    :
```

NOTES:

A) If the GLOBAL identifier is 8 characters including the @ sign there will be a conflict of identifiers at assemble time. To circumvent this situation the following coding technique is recommended:

```
DCL @GLOBLSW   BIT(8) BASED(GLBPTR);

DCL SW         BIT(8) DEF @GLOBLSW;

GLOBX @GLOBLSW (GLBPTR = #R15);
IF X > SW THEN.......;
```

As shown above SW will have the same base address as @GLOBLSW.

B) Values outside the global area can no longer be referenced after using the GLMOD macro (where the protection key is altered).

6) Generally, register contents passed as parameters contain **BINARY** data or address data. If the data type of the field being loaded is arithmetic then the data will be converted to **BINARY**. If the data type is character-string of four characters or less then the character move and blank-fill operations are performed.

7) If a macro requires processing by the compiler but has keyword parameters, enclose each keyword parameter in single quotes:

```
:   ROUTC 'list=R2','lev=d2'(#R2=listptr);     :
:         ...                                  :
:   SERRC 'R,009600,,cp,,'(#R2=myptr);         :
:   GFSCC DB',,'CR;                            :
:                                              :
```

The compiler eliminates the single quotes from the unformatted output statement that is passed to the assembler:

```
:   l      R2,listptr$(R7)      :
:   ROUTC  list=#R2,LEV=d2      :
:   GFSCC  DB,,CR;              :
:                               :
```

It should be noted that although the macro is syntax-checked, the items within the single quotes are passed unchecked.

8) All TPF macros are reserved words. They may not be used for any purpose except to perform the intended operation associated with that word.

9) Macro arguments are passed through the syntax checker; therefore, arguments that do not conform to grammar rules will produce syntax errors:

```
:   SERRC R,'12ABCD'    :
:                       :
```

In the above macro (SERRC) statement, the second positional argument must specify an error message number, consisting only of the hexadecimal numerics 0 through 9 and A through F

10) SABRETALK has a facility to allow macro arguments to bypass syntax checking. This facility should only be used for macro statements that do not require any processing by the compiler (such as those that do not manipulate registers or check for error exits). When an installation's macro table is set up, macros are pre-defined as syntax-checked or not. The programmer has no control over this through a compile.

In effect, the macro statement will be sent through the compiler and the arguments added only when the Assembler Language instructions are generated. Programmers must note that when using non-syntax checked macros that everything will be taken as a parameter until the semicolon delimiter is found. This facility can be used when an installation writes new system macros that do not conform to the TPF positional standard. This should not be interpreted as support for application macros.

To use this facility with a macro, the macro must be added to a special macro table by executing the utility UPDATMAC.

11) The TPF macro GLMOD may be used by applications programs that need to modify data in a protected global area. The storage protection key in the current PSW for the application program concerned will be changed to match that of the global area. The storage protection key must be restored after core modification, generally through the use of the FILKW macro. Between issuing GLMOD and FILKW, the applications program must not issue any ENTER / WAIT type macros, nor should it store data in working storage (**AUTOMATIC**, BASED or ENTRYBLOCK (ECB)).

In order to avert a program interrupt (protection exception), care must be exercised that the statement(s) used to modify the protected area do not reference data that requires either alignment or conversion. Both instances necessitate the use of **AUTOMATIC** storage.

An Example of the Use of the GLMOD Macro:

```
Statement #   Statement
-----------   ------------------------------------

 1            DCL dec5v0 DEC(5) ;
 2            DCL temp1 BIN(31) ALIGNED ;
 3            DCL global1 BIN(31) BASED (gbl1ptr) ;
 4            GLOBX global1(gbl1ptr=#R15);
      ...
 11           GLMOD ;
 12           global1 = dec5v0 ; /* pgm protection exception */
                                 /* occurs here              */
 13           FILKW R ;
```

An alternate coding technique:

```
 20           temp1 = dec5v0 ;
 21           GLMOD ;
 22           global1 = temp1 ;
 23           FILKW R ;
```

The Assembler language coding produced for the statements above would be:

```
Statement #   BAL coding
-----------   ----------------------------------
   12         ZAP   $TEMPDBL(8,R7),DEC5V0$(3,R7)
              CVB   R14,$TEMPDBL(R7)
              ST    R14,GLOBAL1(R6)

   20         ZAP   $TEMPDBL(8,R7),DEC5V0$(3,R7)
              CVB   R14,$TEMPDBL(R7)
              ST    R14,TEMP1$(R7)

   22         MVC   GLOBAL1$(4,R6),TEMP1$(R7)
```

REGISTER LOADING AND STORING

In order to interface with TPF segments and other program segments in the system, it is sometimes necessary to pass data via registers. The loading and storing of registers using values declared in SABRETALK programs is accomplished in conjunction with the **START** statement or with a TPF macro statement.

A) To load registers:

```
TPF macro name  argument(s)  ( ( #reg = | variable | ) )  ;
                             (          | constant | ) )
                             (          | literal  |   )
                             (          -          -   )
                             (                         )
```

B) To store registers:

```
TPF macro name   argument(s)  ( ( variable = #reg ) )     ;
                              (                     )
```

Edited character-strings cannot be referenced in register loading and unloading. The only built-in functions that may be used are the pseudo-variable built-in functions **BSTR**, **CSTR**, and **NSTR**.

C) To load and store registers:

```
TPF     argument(s)   ( ( #reg= | variable | , variable =#reg ) )  ;
macro                 (         | constant |                     )
name                  (         | literal  |                     )
                      (         -          -                     )
```

reg specifies the register to be loaded or stored and may be either the physical register number or its corresponding system-equate.

The following registers are available for use:

Physical number	System equate
0	RAC / RG0 / R0
1	RG1 / R1
2	RGA / RG2 / R2
3	RGB / RG3 / R3
4	RGC / RG4 / R3
5	RGD / RG5 / R5
6	RGE / RG6 / R6
7	RGF / RG7 / R7
14	RDA / R14
15	RDB / R15

Examples:

```
:   START (fare=#R0,cptr=#R7);                  :
:   VAL = BSTR(CMSN,9,8);                       :
:   ENTRC ssb2(#R7=BSTR(VAL));                  :
:   ENTRC ssb4(#R1=aptr,#R2=cmsw,#R0=k,         :
:              #R3=htod,#R4=cptr,#R5=mtod);     :
:   ENTRC ssb3(#R1=answ,cmsw=#R0,sw=#R5);       :
:   BACKC (#R5=cmsw,#R0=k);                     :
:                                               :
```

System Equates in Macro Statements

The number, or hash, sign (#) is also used as the first character of all system-equates:

```
:   v = #PNDRI;   :
:                 :
```

The above will cause the value equated to **#PNDRI** to be stored in v. A # as part of a system-equate and as part of a register parameter may be used in the same statement:

```
:   ENTRC face (#R0 = ordno, #R6 = #SSIREC,    :
:               #R7 = address, error = #R0);   :
:                                              :
```

The compiler resolves system-equates and treats them as fullword **BINARY** literals.

General Rules for Loading and Storing

1) In a **START** statement registers only may be stored.

2) Registers 14 and 15, per TPF standards, may not be used to pass data between programs. They are used only with certain macros (such as CREDC, **GLOBX**, etc.) where they are expected to contain loaded parameters or return values.

3) Subscripting and pointer qualification is allowed for variables to be loaded or stored.

4) Variables may be of the following data types: *NCS, *DEC, *BIN, *BIT and *PTR. If the data type is *NCS or *DEC, the value in the variable will be converted to **BINARY**: data may be lost either because the entire value will not fit in a register, or because fractional data will be lost on conversion of *NCS or *DEC data to **BINARY**. In such cases, the data should be passed via the ECB.

5) Character-string variables may be loaded into registers if they are not more than four characters in length.

Sample Application Supported Macros

Each Installation, according to its needs, will support certain macros. The following are sample Application supported macros.

Name	Function
BACKC	Return control to the previous program block that last issued an ENTR in association with this ECB. Release the automatic storage block associated with the program issuing the BACKC.
CINFC	Allow the application program to read from or write to low, protected core storage and update Control Program Keypoint records. The use of the CINFC macro in SABRETALK is limited to requesting an update of the file copy of Control Program keypoint records, or to obtaining the address of an interface point that contains data.
CONKC	Provide access to a table of system configuration-dependent constants thereby allowing the program's logic to vary, depending on the configuration. The CONKC macro requires a register as its third parameter. The code required by TPF must be followed by the register assignment for SABRETALK For example: **CONKC arg1,arg2,#R0(myarg=#R0);**
CRASC	Send a message to the 1052.
CREDC	Create an independent ECB for deferred processing.
CREIC	Create, for processing, an independent Entry Control Block with the same priority as a new input message.
CREMC	Create an independent ECB for immediate processing.
CRETC	Create an independent ECB and activate a program at a specified later interval of time.
CREXC	Create an independent ECB for low priority deferred processing.
***CRUSA**	Release core block level(s) if attached, or test core block level(s) for states. Address(es) for return of control may be specified.
CSERA	Direct the System Error Routine to take a selective core dump. (CIT and CAC systems).
DEFRC	Defer processing of an ECB until the systems' activity is sufficiently low to allow for the completion of this low priority task.
DLAYC	Delay the processing of the current ECB.
ENTDC	Enter a program and release all program blocks and **AUTOMATIC** storage blocks currently held by the ECB.
ENTNC	Enter a program and release the current active program block and the associated **AUTOMATIC**

Name	Function
	storage block.
ENTRC	Enter a program with expected return.
EXITC	Release the ECB, all associated working storage, **AUTOMATIC** storage blocks, and program blocks, thus terminating the life of the entry in the system.
FILEC	Write a record to file from core storage.
FILKW	Write a keypoint record to file from core storage and retain the core block containing the records' image.
FILNC	Write a record to file from core storage and retain the core block containing the records' image.
FILSC	Write a record, from core storage, to either the prime or duplicate file record.
FILUC	Write a record to file from core storage and 'unhold' the record address.
FINDC	Read a record from file into core storage.
FINHC	Read a record from file into core storage and 'hold' the record address.
FINSC	Read either the prime or duplicate record from file into core storage.
FINWC	Read a record from file into core storage and return control to the operational program when all operations which cause CE1IOC to be incremented, are complete for the ECB.
FIWHC	Read a record from file into core storage, 'hold' the record address and return control to the operational program when CE1IOC is zero.
FLGFC	Write a record from core storage to the next available record in the appropriate data set of the Users' Real-time Disk General File.
FLIPC	Interchange the control information (FARW and CBRW) of two levels.
GDSNC	Initialize a user's File Address Reference Word with the data necessary to access a general data set or a Volume of a general data set.
GDSRC	Convert a relative record number into a File Address (CCHR format) to access a general data set record.
GETCC	Obtain a block of core storage.
GETFC	Obtain an available file storage record address from the specified short or long term pool.
GFSCC	Initiate the Get File Storage (GFS) function which is used to start and stop GFS activity logging, to update the core and file images of all File Pool Keypoint Records, and to start and stop the tagging of long term pool addresses returned to the GFS program via the RELFC macro.
GIVLC	1. Detach a message from a specified level and activate a new ECB with the specified message block attached to level zero of the new ECB. 2. Or send a message via a link with a response expected.
GLMOD	Change the storage protection key in the current PSW, for the application program, to match that of the Global Area, or core resident data records or core resident tables.
GLOBW	Define global fields in global areas 1 and 3.
GLOBX	Define global fields in global area 1.
GLOBY	Define global fields in global area 4.
GLOUC	Prepare the keypoint records for update of the file- resident copy.
KEYCC	Allow for modification of a protected core storage area by an application program by changing the storage protection key in the current PSW.
KEYRC	Restore the current PSW protection key to its normal value.
KEYUC	Prepare the keypoint records for update of the file resident copy.
LMONC	Allow an application program to change the operating state of the CPU from supervisor to problem state.
MONTC	Allow an application program to change the operating state of the CPU from problem to supervisor state.
RCRFC	Release a block of core storage and return a file address to the appropriate pool.
RCUNC	Release a block of core storage and 'unhold' a file record address.
RELCC	Release a block of core storage.
RELFC	Return a file address to the appropriate pool.

Name	Function
ROUTC	Route a data message to a terminal or to an application.
SENDC	Send a message to a terminal.
SERRC	Direct the System Error Routine to take a selective core dump and send a message to CRAS.
TASNC	Assign the previously reserved general tape (TRSV) to the ECB issuing the macro.
TBSPC	Backspace a general tape a specified number of physical records and 'wait' until all operations that cause CE1IOC to be incremented are complete for this ECB.
TCLSC	Close a general tape.
TDFRC	Suspend processing of this ECB for a specified time.
TDLYC	Suspend processing of this ECB for a specified time.
TDTAC	Address a general tape and write from or read into a core storage area.
TOPNC	Open a general tape.
TOURC	Write a record from a core storage block to a real-time tape.
TOUTC	Write a record from core storage to a real-time tape.
TPRDC	Read the next record from a general tape, into core storage.
TREWC	Rewind a general tape and 'wait' until all input operations, including this request, are complete for this ECB.
TRSVC	Reserve a general tape and retain its current positioning until some future ECB takes control with an assign (TASN).
TWRTC	Write a record from core storage to a general tape.
UNFRC	'Unhold' a file storage record address.
WAITC	Defer processing of this entry until CE1IOC is zero.
WRTDC	Write a Critical Data Record from a core block to a user's primary real-time output tape and to the next available record in the appropriate data set of the user's General File.

- Macro parameters are not syntax checked by the compiler.
- For information on Special and User macros see Appendix B

PROCEDURE STATEMENTS

The **PROCEDURE** statement designates the beginning of an external **PROCEDURE**, internal **PROCEDURE**, or programmer-declared function. When coded at the beginning of an internal **PROCEDURE** or programmer-declared function, an optional parameter list may be coded specifying the data being received. External procedures are invoked by coding the **PROCEDURE** label in an enter-type macro, while internal procedures and programmer-declared functions are invoked by coding their labels within the external **PROCEDURE**.

General Format:

A) For external procedures:

```
label:  | PROC      |  ;
        | PROCEDURE |
        -           -
```

B) For internal procedures or programmer declared functions:

```
label:  | PROC      |  (  (  parm1  (,parm2,...,parmn)  )  )   ;
        | PROCEDURE |  (            (                  )     )
        -           -  (            (                  )     )
                       (                                     )
```

(where each parm (parameter) is a variable or a structure)

```
:   extproc: PROCEDURE;                :
:   intproc: PROC (param1, param2) ;   :
:                                      :
```

General Rules for PROCEDURE Statements:

1) Identifiers used as parameters must be declared.

2) A parameter may be one of the following:

   a) variable
   b) structure
   c) array

3) A parameter may not be subscripted nor pointer qualified.

4) When a **PROCEDURE** (or declared function) is invoked, the number of parameters passed to it must be equal to or less than the number of parameters it is to receive. A one-to-one correspondence is established, on a left-to-right basis, between the parameters passed and those received.

5) The **PROCEDURE** statement that begins the program (external **PROCEDURE**) has no parameters. The **START** statement is used to specify parameters the external **PROCEDURE** is to receive via registers.

6) The label on the external **PROCEDURE** statement should follow TPF programming standards.

START STATEMENTS

The **START** statement is used to store parameters passed in registers between programs (See Register Loading and Storing, this chapter).

If an external procedure is receiving parameters from the program invoking it (the invoking program may be a SABRETALK program or an Assembler Language program), the **START** statement should be coded as the first executable statement in the program. (**DECLARE** statements and the **PROCEDURE** statement heading the external **PROCEDURE** are non-executable statements.) All parameters should be stored in a single **START** statement. If the external **PROCEDURE** is not receiving parameters from the invoking program, the **START** statement is not necessary.

General Format:

```
START ( element variable=#reg (,element variable=#reg ) ... )   ;
                              (                        )

      ( reg may be: R0,R1,R2,R3,R4,R5,R6,R7)

   :   START (inptr=#R0,kount=#R4,max=#R5) ;   :
   :                                           :
```

General Rules for START Statements:

1) The **START** statement cannot be used for loading registers.

2) The **START** statement may be used for storing the contents of a register into a variable or pseudo-variable.

3) The registers that may be used for storing are #R0, #R1 #R2, #R3, #R4, #R5, #R6 and #R7.

END STATEMENTS

An **END** statement is used to physically terminate an internal procedure, an external procedure, a programmer-declared function, a **DO** group, and, at times, two or more of the above.

General Format:

```
END ( label ) ;
    (       )

:   END cvbado;   :
:   END error;    :
:   END;          :
:                 :
```

General Rules for END Statements:

1) If a label operand follows the **END**, it cannot be an element of an array of label variables; that is, it cannot be subscripted.

2) If a label operand does not follow the **END**, the statement terminates the last previous unterminated internal **PROCEDURE**, programmer-declared **FUNCTION** or **DO** group. If the unlabeled **END** statement is located as the last source statement of the program then it terminates the external **PROCEDURE**.

3) If the **END** statement terminates a programmer-declared **FUNCTION**, it must not be followed by a label operand.

4) If a label operand follows the **END**, the statement terminates the internal **PROCEDURE** or **DO** group specified; it also terminates all unterminated **DO** groups physically within that **DO** group or **PROCEDURE**. If a label operand follows the **END** and it is located as the last source statement of the program, then it terminates the external **PROCEDURE**. The **END** statement label of an internal or of an external **PROCEDURE** must be the same as the label of its corresponding **PROCEDURE** statement.

```
:   sag1s1:   PROC;          :
:             ...            :
:             exitc;         :
:             ...            :
:   int1:     PROC (a,b);    :
:             ...            :
:             RETURN;        :
:             END sag1s1;    :
:                            :
```

The **END** statement above serves to terminate the internal **PROCEDURE** labelled **int1** and the external **PROCEDURE** labelled **sag1s1**. For the sake of program clarity each **PROCEDURE**, programmer-declared function or **DO** group should have its own **END** statement.

CALL STATEMENTS

The **CALL** statement is used to invoke internal procedures and causes a transfer of control to the specified procedure. An optional parameter list may be coded in conjunction with the **CALL** statement making it possible to pass parameters to the internal procedure.

General Format:

```
CALL  internal_proc_label ( (arg1 (,arg2,...,argn) ) )  ;
                          (       (              )   )
                          (                          )
```

(where each **arg** (argument) is an expression or a structure)

```
:   CALL wrapup;                      :
:   CALL error (a);                   :
:   CALL calc (a+1, CSTR(x,i-5,3));   :
:                                     :
```

General Rules for CALL Statements:

1) The internal **PROCEDURE** label represents the name of the **PROCEDURE** that is being invoked.

2) A **CALL** parameter may be one of the following:

   a) literal
   b) constant
   c) variable
   d) structure
   e) array name (Note: first element only)
   f) any type expression
   g) any type function invocation

3) A parameter may be subscripted and/or pointer qualified.

4) The attributes of a parameter in a **CALL** statement should be compatible with that of its corresponding parameter in the **PROCEDURE** statement, i.e., they must obey the rules for legal assignments.

RETURN STATEMENTS

The **RETURN** statement is used to **RETURN** control from the logical end of internal procedures and programmer-declared functions.

General Format:

A) From an internal procedure:

```
RETURN;
```

B) From a programmer-declared function:

```
RETURN (parameter);
```

```
    :   RETURN;              :
    :   RETURN (a(i));       :
    :   RETURN (ADDR(x));    :
    :                        :
```

General Rules for the RETURN Statement:

1) If a parameter is not specified, the **RETURN** statement must terminate an internal procedure. When such a statement is executed, control is returned to the next sequential statement following the point of invocation.

2) If a parameter is specified, the **RETURN** statement must terminate a programmer-declared function. When such a statement is executed, control is returned to the function invoking statement; the value returned is the value of the parameter, which is used in the completion of the invoking statement.

3) A **RETURN**-statement parameter may be one of the following:

   a) literal
   b) constant
   c) variable
   d) structure
   e) array name (Note: first element only)
   f) any type expression
   g) any type function invocation

4) A parameter may be subscripted and/or pointer qualified.

PROGRAM STRUCTURE

The TPF system is composed of many individual programs, some written in SABRETALK and some written in Assembler Language. This section will describe the program-relative relationship of a SABRETALK program to various other programs or parts of programs.

Rules Governing Program Structure:

A) A SABRETALK program consists of a main body (The MAIN procedure) and optional internal procedures.

B) The names of programmer-declared functions must be declared with the **FUNCTION** attribute.

C) A **START** statement must precede the executable statements if the program is receiving parameters via registers. A **START** statement is the only statement that can be used to receive parameters via registers from other program segments.

D) The first statement in a program is a **PROC** (**PROCEDURE**) statement with no parameters. Its label defines the name of the program. The last statement in a program must be an **END** statement. These two statements serve to mark the beginning and end of a program.

E) Control can be given to another program through the use of a TPF macro such as ENTRC. Whenever control is received from another program via ENTRC, it can be given back through the use of the TPF macro, BACKC.

Using Procedures Compiled Within the Program

Corresponding to the ENTRC and BACKC macros, which provide communication between programs, two statements are provided which allow the use of internal procedures compiled within the main body: the **CALL** statement and the **RETURN** statement.

Figure 5.3 Illustrates the general organization of a program.

```
|PROCEDURE ---|  sub1x1:  PROC;                                          |
|statement     ---                                                      |
|              ---                                                      |
|              |          DCL 1 prog_temps ALIGNED,                     |
|              |                2 (i, j, k, l) BIN,                     |
|              |                2 (p, q, r, s) PTR,                     |
|              |                2 a CHAR(10),                           |
|              |                2 (b, x, z) BIN(31),                    |
|              |                2 c BIN(15);                            |
|              |          DCL 1 record BASED(e) ALIGNED;                |
|              |                2 link ptr,                             |
|              |                2 key,                                  |
|DECLARE -----|                  3 sub_key_1 CHAR(5),                  |
|statements   |                  3 sub_key_2 BIN(31),                  |
|              |                  3 sub_key_3 CHAR(4),                  |
|              |                2 counts,                               |
|              |                  3 total_sold BIN(31),                 |
|              |                  3 dollar_amt DEC(11,2);               |
|              |          %INCLUDEAF EB0EB;                             |
|              |          DCL calc FUNCTION;                            |
|              |          DCL 1 inputs BASED(f),                        |
|              |                2 text CHAR(30),                        |
|              |                2 type BIT(8);                          |
|              ---                                                      |
|              --                                                       |
|START -------|          START (e=#R3); /* base of record in #3 */      |
|statement     ---       ...                                           |
|                        ...                                           |
|              ---                                                      |
|              |          ENTRC ssa6(#R4=q,f=#R3);                      |
|              |          ...                                          |
|              |          CALL look_up(sub_key_3);                     |
|main --------|          ...                                          |
|body          |          ...                                          |
|              |          x = j + calc(q, z - total_sold, 25);          |
|              |          ...                                          |
|              |          EXITC;                                       |
|              ---                                                      |
|              |  look_up: PROC(a);  /* internal PROCEDURE */          |
|              |          ...                                          |
|internal ----|          ...                                          |
|PROCEDURE     |          RETURN;  /* rtrn from internal PROCEDURE */  |
|              |          END look_up;                                 |
|              ---                                                      |
|              |  calc:    PROC(r, b, c);  /* declared function */      |
|              |          ...                                          |
|programmer---|          ...                                          |
|declared      |          RETURN(k + 1); /* return from programmer- */ |
|function      |                         /* declared function       */ |
|              |          END;                                         |
|              ---                                                      |
|              |          BACKC(#R5=k);                                |
|END ---------|          END sub1x1:   /* end of program */           |
|statement     ---                                                      |
|_______________________________________________________________________|
```

Figure 5.3 General Organization of a SABRETALK Program.

CHAPTER 6: ALTERNATIVE CODING METHODS

EFFICIENT PERFORMANCE

SABRETALK allows the programmer a high degree of flexibility to perform operations. Various data types may be combined within statements with all necessary conversions, packing, and alignment handled by the compiler.

There are many methods to solve a given problem. Some methods are more efficient than others are; that is, they accomplish the same result while using fewer resources. In some instances, the programmer must consider the trade-offs that will result from using one method as opposed to another. Use of the ALIGNED or PACKED attribute is an example of this. Use of the ALIGNED attribute may cause unused core between data items, but will result in more efficient code generation. The PACKED attribute will optimize core usage but may result in code inefficiency due to misalignment of data items. It is up to the individual programmer to decide which advantage is more favorable to his particular situation and code his program accordingly.

The following sections outline several areas where the programmer may be able to optimize his program.

DATA CONVERSION

The variety of data types handled by SABRETALK allows the programmer to operate on data in several different methods or to mix various data types in expressions. It is possible to create a situation where unnecessary conversions are required thereby causing a loss of efficiency due to the extra code generated.

When the programmer designs his program and is deciding on the data types for his input data, output data, and work areas, he will be free to use any data type he desires. In certain instances he may be required to **DECLARE** his data to be a certain type due to overall system design. Because of this, he should ask himself the following questions when declaring the remainder of his data:

1. Is a conversion implied? Is the result of an operation assigned to a variable of a different data type?
2. Could the conversion be avoided?
3. If the conversion is within a loop, could it be moved outside of the loop?

The assembler language code generated by the compiler will be kept to a minimum if the following guidelines for the use of *NCS, *DEC, and *BS data are observed:

Use of Numeric Character-String (*NCS) Data

The programmer should recognize the inefficiencies that exist with the use of Numeric character-string data.

*NCS data items are always stored in character (Zoned Decimal) format, one character to a byte. In Zoned format, bytes form a field of four-bit portions as shown:

```
-----------------------------------------------
!    !    !    !    !    !    !    !    !
! Z  ! N  ! Z  ! N  ! Z  ! N  ! Z/S ! N  !
-----------------------------------------------
```

where N specifies a numeric code, Z specifies a zone code, Z/S specifies either a zone or a sign code. The value -234 (shown in hexadecimal) would appear as:

```
F   0   F   2   F   3   D   4
```

where the hexadecimal 'F' is the standard zone code, the hexadecimal 'C' is the standard plus code, and the hexadecimal 'D' is the standard minus code.

Before arithmetic operations may be performed on *NCS data items, they must be converted to the Packed Decimal format. In Packed format, bytes form a field of four-bit portions as shown:

```
-----------------------------------------------
!    !    !    !    !    !    !    !    !
! D  ! D  ! D  ! D  ! D  ! D  ! D  ! S  !
-----------------------------------------------
```

where D signifies a decimal digit, S signifies a sign code. The value -234 (shown in hexadecimal) would appear as:

```
0   0   0   0   2   3   4   D
```

If data items must be declared as *NCS to conform to input or output requirements, and if they will also be used in arithmetic calculations, they should be converted to another data type prior to the arithmetic operation. They should not be converted back to *NCS until all arithmetic operations have been completed.

Use of Decimal Data

If fractional values are unnecessary and arithmetic is to be performed, it is much more efficient (both in storage requirements and execution time) to **DECLARE** it as **BINARY** rather than **DECIMAL**. Since arithmetic operations are frequently performed in **DECIMAL**, consideration should be given to declaring operands of arithmetic expressions as **DECIMAL**.

```
:   DCL (a, d) BIN(15), b DEC(3,0);   :
:   a = b * d + 5;                    :
:                                     :
```

In the above example, the operation **b*d** will be performed first; **d** will be converted to **DECIMAL**, then the multiplication performed. The **BINARY** literal 5 will then be converted to **DECIMAL** and added to the intermediate result of **b*d**. This final result will be in **DECIMAL** which will then be converted to **BINARY** for storage into **a**.

```
:   DCL (a, d) BIN(15), b DEC(3,0), c(10) CHAR(4);   :
:   c(2 * b) = 'a';                                  :
:                                                    :
```

In the above example, the literal 2 will be converted to **DECIMAL** and, after multiplication, the result converted back to **BINARY**.

Better code will result if all **DECIMAL** fields are declared with an odd number of digits. This is because, on the IBM System/360, **DECIMAL** fields are stored in the packed decimal format and the sign occupies the last position of the last byte. If possible, all **DECIMAL** variables should be declared with the same number of digits in the fractional portion, so as to further optimize the generated code.

Use of BIT-String Data

Under the following circumstances, the use of **BIT**-strings in arithmetic expressions is as efficient as **BINARY**:

1) The length of the string is 8 bits and byte aligned.
2) The length of the string is 16 bits and halfword aligned.
3) The length of the string is 32 bits and fullword aligned.

Much more code is generated for **BIT**-strings where the length is not a multiple of 8.

AVOIDING POOR PROGRAMMING TECHNIQUES

The results of a relational operation, either a one (1) or a zero (0), may be used in many other types of statements besides the **IF** statement. The value returned may be used in an arithmetic expression, an Assignment statement, a **CALL** statement, a **RETURN** statement, etc.

```
:   CALL intproc( (a < b) + (c = d) + (e ^= f) );   :
:                                                   :
```

In the preceding example, a value of 0, 1, 2 or 3 will be passed to the internal **PROCEDURE** depending upon how many of the relational (comparison) operations held true.

Compiler Restriction

The compiler limit that is restrictive is the limit of 255 on dimensions. In certain situations this limit can easily be circumvented. The following circumvention is not to be used if the area is a core block attached to the ECB, whose limit (127,381,1055,etc.) may be violated.

If a dimension larger than 255 is required, the user can specify an area following the declared area. Indexing can then proceed through both areas.

```
:   DCL a(200) CHAR(3);     :
:   DCL a2(100) CHAR(3);    :
:   DCL (i, j) BIN;         :
:       ...                 :
:   j = 0;                  :
:   DO i = 1 TO 300;        :
:      IF a(i) = 'jfk'      :
:        THEN j = j + 1;    :
:   END;                    :
:                           :
```

The preceding example will count the occurrences of the character-string 'jfk' in a 300-element array. The declaration of **a2** is really an extension of **a**. Indexing will proceed through all 300 entries with no problem. When extending an array in this manner, the second declaration (**a2** in the example) should immediately

follow the first. Note that if **a** were in **BASED** storage (say a 1055-byte block), the same technique can be applied but the **DCL** for **a2** is unnecessary.

```
:   DCL a(200) CHAR(3) BASED(aptr) ;   :
:   aptr = ce1cr2 ;                    :
:   j = 0 ;                            :
:   DO i = 1 TO 300 :                  :
:      IF a(i) = 'jfk'                 :
:        THEN j = j + 1 ;              :
:   END ;                              :
:                                      :
```

Other Techniques For Efficient Coding

```
:   /*  poor method  */                :
:   IF frmtoday < -1 | frmtoday > 1    :
:     THEN GOTO a;                     :
:                                      :
:   /*  better method  */              :
:   IF ABS(frmtoday) > 1               :
:     THEN GOTO a;                     :
:                                      :
```

```
:   /*  poor method  */     :
:   IF seccode = 'd'        :
:     THEN holdsw = '1'b;   :
:   IF seccode = 's'        :
:     THEN holdsw = '1'b;   :
:   IF seccode = 'p'        :
:     THEN holdsw = '1'b;   :
:   IF seccode = 'c'        :
:     THEN holdsw = '1'b;   :
:                           :
:   /*  better method  */   :
:   IF seccode = 'd' |      :
:      seccode = 's' |      :
:      seccode = 'p' |      :
:      seccode = 'c'        :
:     THEN holdsw = '1'b;   :
:                           :
```

In the example above, the second method proves to be more efficient both in generated code and execution time. In the first set of statements, the BAL code to set **holdsw = '1'b** will be generated four times and all four **IF** statements will always be executed.

```
:   /*  poor method  */     :
:   IF pchetc(c) > 1200     :
:     THEN cmram = 'p';     :
:   IF pchetc(c) = 1200     :
:     THEN cmram = 'n';     :
:   IF pchetc(c) = 2400     :
:     THEN cmram = 'm';     :

:   /*  better method  */   :
:   bchetc = pchetc(c);     :
:   IF bchetc > 1200        :
:     THEN cmram = 'p';     :
:   IF bchetc = 1200        :
:     THEN cmram = 'n';     :
:   IF bchetc = 2400        :
:     THEN cmram = 'm';     :
```

In the example above, the second set of statements causes the Assembler Language code necessary to calculate the subscript to only be generated once.

In the following example the value **`place(race)`** is produced ten times as a result of unnecessary subscription within a loop:

```
:          car = 0 ;                        :
:   loop:  car = car + 1 ;                  :
:          team(car) = place(race) ;        :
:          IF car ^= 10 THEN GOTO loop ;    :
:                                           :
```

Move this subscripted value 'place(race)' outside of the loop and many undesirable processing steps will be saved:

```
:          points = place(race) ;           :
:          car = 0 ;                        :
:   loop:  car = car + 1 ;                  :
:          team(car) = points ;             :
:          IF car ^= 10 THEN GOTO loop ;    :
:                                           :
```

A very efficient way to zero a structure that is less than 256 bytes in length is illustrated below:

```
:   DCL 1 input BASED(inptr),            :
:         2 team CHAR(30),               :
:         2 habitat,                     :
:           3 street CHAR(20),           :
:           3 city CHAR(20),             :
:           3 state CHAR(18),            :
:           3 zip PIC'99999',            :
:         2 age PIC'999',                :
:         2 salary DEC(6,2);             :
:                                        :
:   BSTR(input,1,8) = '00'x;             :
:   CSTR(input,2) = CSTR(input);         :
:                                        :
```

An area larger than 256 bytes may be initialized using the same general technique:

```
:   dcl 1 struct BASED (sptr),                    :
:         2  team CHAR(250),                      :
:         2  city CHAR(250),                      :
:         2  zip  CHAR(250),                      :
:         2  age  CHAR(250);                      :
:         2  key  CHAR(55);                       :
:   BSTR(struct,1,8) = '00'x;                     :
:   CSTR(struct,2)   = CSTR(struc);               :
:                                                 :
```

Code generated:

```
: *  BSTR(STRUCT,1,8) = '00'x;                        :
:         MVI   STRUCT$+1(256,R5),X'00'               :
: *  CSTR(STRUCT,2)   = CSTR(STRUC);                  :
:         MVC   STRUCT$+1(256,R5),STRUCT$(R5)         :
:         MVC   STRUCT$+257(256,R5),STRUCT$+256(R5)   :
:         MVC   STRUCT$+513(256,R5),STRUCT$+512(R5)   :
:         MVC   STRUCT$+769(256,R5),STRUCT$+768(R5)   :
:         MVC   STRUCT$+1025(30,R5),STRUCT$+1024(R5)  :
```

Initializing a field

Techniques for initializing a field are illustrated as follows:

```
:   DCL msg CHAR(4);                                          :
:                                                             :
:   msg = '    '    ;  /*  initialize to blanks           */  :
:   msg = '0000'    ;  /* init'lize to zoned decimal zeros */  :
:   BSTR(msg) = '0'b;  /*  initialize to binary zeros      */  :
:                                                             :
```

Testing and setting single-bit fields in the following manner is very efficient.

```
:   DCL (a, b, c) BIT(1);   :
:                           :
```

To set bits to zero or one:

```
:   b = '0'b;   :
:   b = '1'b;   :
:               :
```

To invert a bit:

```
:   /*  poor method  */          :
:   IF b = '1'b THEN b = '0'b;   :
:                                :
:   /*  better method  */        :
:   b = ^b;                      :
:                                :
```

To test bits:

```
:   IF a = '1'b | b = '0'b | c ^= '1'b | d = '0'b THEN x = y;   :
:                                                               :
```

Use of Logical Operations

The following examples illustrate methods in which the logical operations can be used advantageously:

Suppose one wants to place EBCDIC numeric zones (all one bits) into a two-character field, **k**.

```
:   DCL k CHAR (2);                :
:   BSTR(k) = BSTR(k) | 'F0F0'X;   :
```

This operation causes the hex digit **F** (**1111** in binary) to be inserted into the zones of field k. The numeric portions of k are not altered.

Suppose one wants to test for odd or even in a value. A function to test for odd could be as follows:

```
:   DCL odd FUNCTION;   :
:   DCL arg BIN(31);    :
:   odd: PROC(arg1);    :
:   RETURN(arg & 1);    :
:   END;                :
:                       :
```

The expression in the **RETURN** statement causes all bits of **arg** to be set to zero except the rightmost which is zero (false) if the number is even, but one (true) if the number is odd. This function might be used as follows:

```
:   IF odd(i + j) = 1 THEN GOTO x_odd;   :
:                                        :
```

i and **j** are added and if the result returned by odd is one, the program will **GOTO x_odd**.

A different test for odd/even is:

```
:   If MOD(x,2) = 1 THEN GOTO x_even;   :
:                                       :
```

Bit Manipulation

It is seldom necessary for the programmer to use bit manipulation like his assembler language counterpart, but occasions will arise. By judicious use of **&**, **|**, **^**, **SHL** and **SHR**, it is possible to manipulate bits to almost any degree. In most cases this results in the generation of more suitable coding.

To illustrate this, consider the following case: there is a code word (**codewd**) which contains 32 conditions numbered 0 through 31. Each bit in **codewd** stands for a condition. If it is on, the bit is one, if it is off, the bit is zero. At various times it will be necessary to set and test these bits.

First, let us examine how to test whether bits 26, 28 and 30 are on.

```
:   IF (codewd & '101010'b) = '101010'b THEN x = y;   :
:                                                     :
```

The **&** operation "erases" all but the relevant three bits. The result of this is compared to see if the three relevant bits are on. If it is desired to set 'on' the nth-bit-from-the-right (where n=codex=0 to 31) of **codewd**, the following can be used.

```
:   codewd = codewd | SHL(1, codex);   :
:                                      :
```

The **SHL** function positions the single bit specified in the first parameter at the appropriate position specified by codex. The **|** operation then inserts this bit into **codewd**. Several values could be set with one assignment.

```
:   codewd = codewd | SHL(1, codex) | SHL(1, codey);   :
:                                                      :
```

If it is desired to test a particular bit on the basis of codex, the following can be used.

```
:   IF SHR(codewd, codex) & 1 = 1 THEN GOTO sam;   :
:                                                  :
```

The value **codewd** is shifted and then all but the low bit is eliminated by the **&**. If the result is not zero, the program will go to **sam**.

Suppose it is decided to set a bit in **codewd** off on the basis of codex.

```
:   codewd = codewd & (^SHL(1, codex));   :
:                                         :
```

In this case the mask is created and positioned by the **SHL**. However, we want to insert a zero and not a one. All positions of the mask now contain ones except that to be set to zero. The **&** operation will thus leave all bits except for that to be zeroed, set as they were.

A higher level of complexity is to set a relevant bit in **codewd** to a value contained in the bit named **x**.

```
:   codewd = (codewd & ^SHL(1, codex)) | SHL(x, codex);   :
:                                                         :
```

This first causes the appropriate bit in **codewd** to be set to zero. The bit **x** is then positioned by the second **SHL** and inserted into **codewd** with the **|** operation. Note that **x** may contain zero or one.

Several other notes on bit manipulation are relevant:

- It is possible to concatenate **BIT**-strings.

- Watch out for negative numbers. For example, a SHL of a positive number will result in a negative number if the sign bit has a one in it after the shift.

- The effect of algebraic shifting can be obtained by multiplying or dividing by a power of 2.

- The logical operators and the SHL/SHR functions can be used on data declared as **BINARY** or on expressions.

- If a **BIT** field is not aligned on a byte boundary, it may be better to move it to a work area before performing many calculations. Then, when processing is complete, move it back.

- Use comments to describe your program logic, data variables, and input and output fields.

- While it is legal to write an expression in an **IF** statement with no relational operator (See Chapter 3, **IF** Statement), it often turns out that an explicit comparison with zero results in better code generated.

```
:   /*  poor method  */     :
:   IF ^a THEN GOTO c;       :
:                            :

:   /*  better method  */    :
:   IF a = 0 THEN GOTO c;    :
:                            :
```

Figure 6.1 is an example of a SABRETALK program:

```
    exa4d1:  PROC;
             %INCLUDEAF  eb0eb;

             DCL  1  workarea,
                     2  prntblk,
                        3  header   CHAR(12),
                        3  text     CHAR(20),
                     2  message,
                        3  id        DEC(3),
                        3  msgnbr    DEC(5,2),
                        3  msgcode1  BIT(8),
                        3  msgcode2  BIT(3),
                        3  msgcode3  BIT(29),
                        3  msg       CHAR(32);

             DCL  1  fltinfo         BASED(fptr),
                     2  fltno(2)     BIN(31),
                     2  fltin        BIN(15),
                     2  fltout       BIN,
                     2  flttbl(3)    CHAR(8);
             DCL  1  tbl1(8)         DEFINED tbl2,
                     2  seat         BIN(15),
                     2  meal         CHAR(3);

             DCL  tbl2(8)            BIN(15) ALIGNED CONSTANT;
             DCL  (ptr1,ptr2) PTR    ALIGNED;
             DCL  find               BIN(31) ALIGNED;
             DCL  lblsw(3) LAB;
             DCL  days               BIN;
             DCL  hours              BIN;
             DCL  minutes            BIN;
             DCL  seconds            BIN;
             DCL  (a,b,c,d,e,f)      BIN ALIGNED;
             DCL  (i,j,k)            BIN;

             CONST tbl2, '0001001000110100'B ;

     entry1: START(fptr=#R2,f=#R3);
             ptr2 = addrtbl2;
             ENTRC exa5d1(#R3=ptr2,find=#5);
             CSTR(ebw021,1,4)=VSTR(ebw001,1,k);
             GOTO lbl4;
    tg2:     IF find = 0 THEN GOTO err;
    err:     CSTR(ebw021,j)='john';
    dun:     ENTNC exa6d1;
    beginlbl: a=b+c-d*e/f;
             lblsw(1)=fptr;
             i,f=0;                  /* clear loop controls */
             k='10100'B ;
             d=16 ;
```

```
            DO  I=1 TO 16 BY 3 WHILE K < A;
                a=a+1;
                IF b > 7 THEN GOTO lbl4;
                ELSE
                  DO  a=a-1;
                      b=b+1;
                  END ;
                IF c ^= d THEN
                  DO  f=d+e;
                      k=b+a;
                  END ;
                ELSE a=k+i;
            END ;

            c = '_         F8_       C_ D9_ A3'X ;
            msgnbr = i+k;
  lbl4:     days = d-a;
            msg = text || flttbl;
            GOTO extproc;
  dcldfunc: PROC(a);
            RETURN(c+3);
            END ;
  intproc:  PROC(a);
          ......
            RETURN;
            END intproc;

  extproc:  ENTRC ssa4d1(#R0=b);
            BACKC(#R3=fptr);
            END exa4d1;
```

Figure 6.1 Example of a SABRETALK program:

CHAPTER 7: SABRETALK COMPILER OPTIONS

The output of the compiler is determined by pre-established installation default parameters. These defaults are initialized by the systems programmer but may be temporarily overridden by the programmer.

The compiler options established as defaults may be overridden temporarily by the programmer, for his compilation, through the use of the loader **OPTIONS** statement.

OPTIONS STATEMENTS

The format of the **OPTIONS** statement is as follows:

```
Card column 1----|
                 |
                 V
                 OPTIONS=aaaaa(,bbbbbbb...)
                              (           )
```

The **OPTIONS** keyword must begin in column one. The next non-blank following the keyword must be an equal sign. The next non-blank following the equal sign must be one or more compiler options, each of which is separated from the preceding one by a comma. This statement may be coded through column 71. In the event that all selected options will not fit on one statement, multiple OPTIONS statements must be used. Notice that there is no statement terminator (**;**) for compiler options. Imbedded blanks are not allowed.

```
Card column 1----|
                 |
                 V
                 OPTIONS=INCLD,ICAFNO
                   or
                 OPTIONS=INCLD
                 OPTIONS=ICAFNO
```

All **OPTIONS** statements must immediately precede the PROCEDURE statement for the program.

```
Card column 1----|
                 |
                 V
                 OPTIONS=NOCODE,ONL
                  abcda0: PROC;
                     ...
                  END abcda0;
```

Details on temporary overrides of compiler options, a list of compiler options and a description of each of the compiler options follows. The programmer should be sure to check which of these options are installation defaults.

Sample Options List: (* indicates suggested defaults)

```
  * ALIGN               NOALIGN
  * ALPHA
  * ANGB3               NOANGB3
  * BAL                 NOBAL
    CLEAR             * NOCLEAR
  * CODE                NOCODE
  * DECK                NODECK
  * DOLLAR              NODOLR
    GEN               * NOGEN
  * ICAFYES             ICAFNO
    INCLD             * NOINCLD
  * MAP                 NOMAP
  * MLEVEL0             MLEVEL1             MLEVEL2
  * NUMERIC
    OPT               * NOOPT
  * PRINT               NOPRINT
  * SPACE
    System Equate Identifiers    (* GTS)
    TERM              * NOTERM
  * XREF                NOXREF
```

ALIGN / NOALIGN

The NOALIGN option will take advantage of the SYS/370 Byte-Oriented-Operand facility that permits storage operands of most unprivileged instructions to appear on any byte boundary.

If the data item is misaligned it will be flagged as thus, but the compiler will not attempt to align the item, nor will it produce the move instructions to aligned field (temporary field).

The programmer should be also be aware that when the NOALIGN option is in affect some performance degradation is possible when storage operands are not positioned at addresses that are integral multiples of the operand length. It is suggested that fields that appear as misaligned and frequently used should be aligned on integral boundaries.

ALPHA

```
          _       _
ALPHA=(|TBLADDR|,name)
       |TBL    |
       -       -
```

The ALPHA compiler option relates to the translate table the compiler references when generating code for the ALPHA built-in function. It also indicates the manner in which the table is accessed.

name is required. It indicates the identifier the compiler will use when generating code for the ALPHA function. The compiler always generates the GLOBZ macro for the ALPHA function. name is the identifier within the global area that either contains the address of the table or is the name of the table. name is therefore always a global tag.

TBL and TBLADDR are mutually exclusive. One of the two must be specified. TBL indicates to the compiler that the name specified is the name of the

table that is to be used when generating code for the ALPHA function. TBLADDR indicates to the compiler that the name specified is a global field containing the address of the table.

In the case where @trtal contains the table name:

```
:   ALPHA=(TBL,@trtal)       :
:                            :
```

In the case where @trtap contains the table address:

```
:   ALPHA=(TBLADDR,@trtap)   :
:                            :
```

ANGB3 / NOANGB3

This is a system-dependent consideration and should not concern the applications programmer. When the tables for **ALPHA / NUMERIC** built-in functions are relocated from the usual global area 1 to global area 3, and references to global area 3 are desired, then the **ANGB3** option is required. With the **ANGB3** option global macros are able to reference fields in either global area 1 and/or 3. This is because the compiler generates the **GLOBZ** with the **fld=** parameter when **ANGB3** is specified. The macro definition of GLOBZ used by the assembler must be the version that recognizes the **fld=** parameter and its values and which generates appropriate GL1 and GL3 addressability). The **NOANGB3** option allows global references to fields in global area 1 only.

```
:   GLOBX @u1day (dayptr = #R15);   :
:                                   :
```

Code generated:

```
:   GLOBZ REGR=R15,FLD=@u1day   :
:   LA    R15,@u1day            :
:   DROP  R15                   :
:   ST    R15,dayptr$(R7)       :
:                               :

:   positn = ALPHA(field);   :
:                            :
```

Code generated:

```
:   GLOBZ REGR=R15,FLD=@trtal     :
:   LA    R15,@trtal              :
:   DROP  R15                     :
:   SR    R1,R1                   :
:   LA    R14,field$(R7)          :
:   TRT   field$(20,R7),0(R15)    :
:   BE    *+10                    :
:   SR    R1,R14                  :
:   LA    R1,1(,R1)               :
:   STH   R1,positn$(R7)          :
:                                 :
```

BAL / NOBAL

OPTIONS=KEY=password,BAL where password is the directory protection key. This option allows the programmer to code BAL statements within a SABRTALK program by placing an '**X**' or '**.**' in card column one. If **OPTIONS=KEY=password,NOBAL** where password is the directory protection key is in effect and an '**X**' or '**.**' is coded in card column one the following error will be issued:

SBT0042E EMBEDDED BAL STATEMENTS NOT ALLOWED.

CLEAR / NOCLEAR

OPTIONS=CLEAR=(xx) where **xx** is the Hex pad byte. Coding **CLEAR=(00)** is the same as Coding **CLEAR** with no options and will produce code to clear the user portion of the **AUTOSTORAGE** block to hex zero's. Coding **CLEAR=(40)** will clear the **AUTOSTORAGE** block to blanks.

EXAMPLES:

```
1)   CLEAR or CLEAR=(00)
           ALASC L0
           MVI   4(R7),X'00'
           MVC   5(113,R7),4(R7)

1a)  CLEAR=(40)
           ALASC L0
           MVI   4(R7),X'40'
           MVC   5(113,R7),4(R7)

2)   CLEAR or CLEAR=(00)
           ALASC L1
           MVI   4(R7),X'00'
           MVC   5(113,R7),4(R7)
           MVC   118(254,R7),117(R7)

2a)  CLEAR=(40)
           ALASC L1
           MVI   4(R7),X'40'
           MVC   5(113,R7),4(R7)
           MVC   118(254,R7),117(R7)

3)   CLEAR or CLEAR=(00)
           ALASC L2
           STM   R0,R1,4(R7)
           LA    R14,12(,R7)
           LA    R15,1034
           SLR   R1,R1
           MVCL  R14,R0
           LM    R0,R1,4(R7)
           XC    4(8,R7),4(R7)
```

```
3a)   CLEAR=(40)
           ALASC L2
           STM   R0,R1,4(R7)
           LA    R14,12(,R7)
           LA    R15,1034
           SLR   R1,R1
           ICM   R1,B'1000',=X'40'
           MVCL  R14,R0
           LM    R0,R1,4(R7)
           MVI   4(R7),X'40'
           MVC   5(7,R7),4(R7)

4)    CLEAR OR CLEAR=(00)
           ALASC L4
           STM   R0,R1,4(R7)
           LA    R14,8(,R7)
           LA    R15,4075
           SLR   R1,R1
           MVCL  R14,R0
           LM    R0,R1,4(R7)
           XC    4(8,R7),4(R7)

4A)   CLEAR=(40)
           ALASC L4
           STM   R0,R1,4(R7)
           LA    R14,8(,R7)
           LA    R15,4075
           SLR   R1,R1
           ICM   R1,B'1000',=X'40'
           MVCL  R14,R0
           LM    R0,R1,4(R7)
           MVI   4(R7),X'40'
           MVC   5(7,R7),4(R7)
```

CODE / NOCODE

If **CODE** is specified, an Assembler Language source listing will be generated intermixed with the SABRETALK statement source listing. These options are independent of the **DECK / NODECK** options.

DECK/NODECK

If **DECK** is specified, the compiler will produce an Assembler Language source input data set unless a terminal type error occurs. If other errors are noted during the compilation, this data set will still be created; however, the return code will describe the severity of the error.

Return Code	Meaning
0	Compilation successful
4	Information message generated
8	Warning message generated
12	Error message generated
16	NODECK option

The **NODECK** option always produces a return code of 16.

DOLLAR / NODOLR

If **DOLLAR** is specified, the compiler will append dollar signs to every variable less than eight characters in length. The regular **BEGIN** macro will be generated.

If **NODOLR** is specified, dollar signs will not be appended and the **BEGIN** macro generated will be

```
:   BEGIN ECB=NO,NAME=xxxx,VERSION=xx   :
:                                       :
```

If the **NODOLR** option is invoked, the application programmers should be aware of system tags generated by the **BEGIN** and **FINIS** macros, tags (names) that may be duplicates of those generated by SABRETALK. The programmer should also avoid using the standard tags generated by the compiler such as **$LIT, $TEMPxxx, $GENxxxx**.

In cases where the **DOLLAR** compiler option is in effect and the global tag **@fnc00** is less than eight characters, the statement in the following example:

```
:  DCL @fnc00 BIN(31) BASED fonptr ;   :
:                                      :
```

will cause the assembler to generate:

1) **@fonc00$** for the field tag, and
2) **LA RDB,@fnc00** for the expansion instruction.

These are technically (and need to be) two different names. If the global tag is eight characters or more however, no dollar sign is appended: the two names are identical and a "PREVIOUSLY DEFINED NAME" error diagnostic appears.

GEN / NOGEN

This option is used to provide additional information when the **BEGIN** macro is encountered. If **GEN** is specified, the Assembler Language source statement **PRINT GEN** will be added to the Assembler Language source input data set just before the **BEGIN** macro. (The programmer should be aware that the **BEGIN** macro generates a **PRINT NOGEN** as its last statement). The default option **NOGEN** produces **PRINT NOGEN**.

ICAFYES / ICAFNO

This option relates to the method by which data records referenced by **%INCLUDE** are accessed. If a programmer codes a **%INCLUDE** statement, and the **ICAFYES** option is invoked, the compiler will first attempt to access the record from the **%INCLUDEAF** file. If the record is on the **%INCLUDEAF** file, it will be used. If not on the **%INCLUDEAF** file, the record located on the **%INCLUDE** file will be used. If on neither file, an error would be generated.

If the programmer codes a **%INCLUDE** statement and the **ICAFNO** option is invoked, the compiler will look for the record on the **%INCLUDE** file only.

INCLD / NOINCLD

When **INCLD** is specified, a source listing of all statements read in from the **%INCLUDE** file will be generated. These statements are always treated as user-coded compiler statements, regardless of the option specified.

MAP / NOMAP

When **MAP** is specified, the user will receive an Attribute File Map and a Register Usage Map for each compilation. These maps describe the layout of storage and register assignments required by the program.

Attribute file map listings will have the following format:

```
                ATTRIBUTE---FILE---MAP

 LEVEL NAME   BASE   TYPE   HEX    LOC   SIZE DIM SSMULT DEC-DIG CARD#

 1     bbb$   AUTO   BIN    004    4:0     4   1                  2
 1     ccc$   AUTO   BIN    008    8:0     4   1                  2
 1     fea$   AUTO   DEC    00C   12:0     3   1         5.02    11
 1     aaa$   AUTO   CHAR   00F   15:0    20   1                  12
 1     b$     AUTO   BIN    023   35:0     2  10  2              15 misaligned
 1     i$     AUTO   BIN    037   55:0     2   1                 16 misaligned
 1     dd$    AUTO   ECS    039   57:0     4   1          .00    18
 1     ee$    ptr1$  STR           :0    20   1                  29
 1     ptr1$  AUTO   PTR    040   64:0     4   1
 1     ee1$         CHAR           :0    20   1                  30
```

An explanation of each column follows:

- **LEVEL** the structural level of the named item.
- **NAME** the name of the declared item.
- **BASE** the storage class of the item.
- **TYPE** the data type to which the item belongs.
- **LOC** the byte location (displacement) of the item (in decimal), followed, when needed, by the decimal number of the bit (within byte) location.
- **SIZE** the length of the item in bytes and, where needed, in additional bits.
- **DIM** for arrays, the number of items in the array (in decimal) - - otherwise the value 1.
- **SSMULT** the number of bytes per array item in decimal.
- **DEC-DIG** for **DECIMAL** data, the decimal number of integer digits, followed by the decimal number of fraction digits.
- **CARD#** the sequence number of the card record in which the item appears.

Register Usage Map listings will have the following format:

```
                REGISTER---USAGE---MAP

   BASE NAME       STORAGE ALLOCATED     REGISTER ASSIGNMENTS

   AUTOMATIC       544                   reg  7  R7
   CONSTANT                              reg  8  R8
   ENTRY BLOCK                           reg  9  R9
   CONTROL PGM                           reg 10  R10
   CONTROL PGM                           reg 11  R11
   CONTROL PGM                           reg 12  R12
   CONTROL PGM                           reg 13  R13
   rptr$           800                   reg  6  R6
```

MLEVEL0 / MLEVEL1 / MLEVEL2

If **MLEVEL0** is specified, all error, warning and information messages will be listed in the diagnostic listing at the end of each compilation.
MELVEL1 - error and warning messages only.
MLEVEL2 - error messages only.

NUMERIC

```
NUMERIC=(|TBLADDR|,name)
         |TBL    |
         -       -
```

The **NUMERIC** compiler option relates to the translate table the compiler references when generating code for the **NUMERIC** built-in function. It also indicates the manner in which the table is accessed.

name is required. It indicates the identifier the compiler will use when generating code for the **NUMERIC** function. The compiler always generates the **GLOBZ** macro for the **NUMERIC** function. **name** is the identifier within the global area that either contains the address of the table or is the name of the table. **name** is therefore always a global tag.

TBL and **TBLADDR** are mutually exclusive. One of the two must be present. **TBL** indicates to the compiler that the name specified is the name of the table that is to be used when generating code for the **NUMERIC** function. **TBLADDR** indicates to the compiler that the name specified is a global field containing the address of the table.

In the case where **@trtno** contains the table name:

```
:   NUMERIC=(TBL,@trtno)       :
:                              :
```

In the case where **@trtad** contains the table address:

```
:   NUMERIC=(TBLADDR,@trtad)   :
:                              :
```

OPT / NOOPT

If the option **OPT** is invoked, the compiler will optimize the literal pool. If **NOOPT** is invoked, the literal pool optimization pass is omitted. If an error is encountered when the program is compiled, the **NOOPT** option is automatically invoked for that compile.

PRINT / NOPRINT

The PRINT / NOPRINT option is a general control parameter and will override the other print related options: MAP, XREF, CODE, etc. If PRINT is specified, output will be provided in accordance with the other print related options. If NOPRINT is specified, there will be no printed output produced, regardless of the other options.

System-Equate-Identifiers Options (GTS, ONL, etc.)

This option specifies a three-character code designating the group of system-equate tags that will be recognized when compiling the program. An installation can specify up to four three-character tags that relate to the particular system-equate macros (valid within the BDAM file. The program may only refer to one **SYSEQ** at a time). The two indicated above are only examples. The installation establishes the valid three-character tags by executing the utility **RESETEQU**. The default option is established by executing a **PERM** type run. When an installation wishes to change any of the three character tags, within a particular system-equate, the utility **UPDATEQU** must be executed.

TERM / NOTERM

The **TERM** option specifies that diagnostic messages, along with their corresponding source statements, are to be placed in the data set, or the device, described by the **SYSTERM DD** card in the JCL compilation request. This option is similar to the Assembler option of the same name.

TRACE / NOTRACE

The **TRACE** option gives SABRETALK code in the BAL listing.

XREF / NOXREF

If XREF is specified, a cross-reference listing will be provided specifying the statement number of those statements that define or use the symbols of a program. An asterisk following a statement number denotes that the symbol was referenced as a receiving field in that statement.

```
                    CROSS--REFERENCE--LISTING

    SYMBOL      DEFN   REFERENCES

    array2$     138    245   287   287   288
    array3$     139    220   234   240   303   313
    flt_no$     150    432
    highchk$     77    123   127   129   254   254   255   261
                       266   267   269
    ndx4$        48    123*
    tag$         43    327
```

The **SYMBOL** column will contain the alphabetized names of identifiers and labels that are essential in the assembled program. Macro arguments and **INCLUDEAF** identifiers may also appear.

The **DEFN** column will contain the decimal number of the source statement that defines the symbol. The names of macros and the word **INCLUDEAF** may appear as counterparts of **SYMBOL** entries.

The **REFERENCES** column will contain the decimal number(s) of statements wherein the symbol is referenced.

Compiler Support of Variable Block Sizes:

The compiler will accept user-defined size limits for the four storage block classes. Specification of limits will be accomplished through the use of new compiler options.

Option Keyword	Option Meaning	Option Syntax
KEY	PROTECTION KEY	key=password
ECB	ENTRY CONTROL BLOCK	ecb=size
PSB	PROGRAM (CONSTANT) STORAGE BLOCK	psb=size
BSB	BASED STORAGE BLOCK	bsb=size
ASB	AUTOMATIC STORAGE BLOCK	asb=(name1,size1, name2,size2, ...)

password: a one to four character sequence (excluding commas and blanks), controlled by an installation's Compiler Support Group

size: one to four decimal digits, whose value must not exceed 4096 (4k)

name: a one to eight character sequence which should be a valid argument for the **ALASC** macro (compiler does not validate)

Option specifications must have no imbedded blanks, and must be separated by commas.

During compilation (**EXEC**) runs, the correct **KEY=password** specification is required in order to override default block sizes, and it must be coded prior to any block option.

Option overrides may be specified on a SABRETALK **OPTIONS** statement, or by a processor override statement available to your System Control Group.

Example:

```
OPTIONS=KEY=ZZZ,PSB=4096
```

- must begin in column one (1)
- coded prior to the first program statement

All compiler options in effect (with the exception of **KEY**) are listed at ends of compilations. The **KEY** will be listed at the ends of compiler **PERM** runs.

To provide full support for the **PSB** option, the compiler's core management philosophy has been revised. Consequently, it is possible to compile most programs of 4K bytes within the typical region size of 220K, supplied by most SABRETALK users running under standard OS/MVT.

In the event that the available main storage for a compilation is exhausted, however, the following Terminal error message is issued:

```
SBT0186T  -  INSUFFICIENT CORE ALLOCATED FOR THIS COMPILATION.  RUN ABORTED.
```

This unfortunate and typically rare condition can be remedied by increasing the region size used for the compilation.

Word of Caution:

These options should not be used indiscriminately. It should be understood that they only provide compilation flexibility, and in no way insure that a resultant program will be compatible with attributes of a given TPF System. At System One, for example, the TPF modifications required to support core-

resident programs up to 4K bytes in size (PSB=4096), consisted of changes to the Loader, the Allocator, the Post Processor and the Control Program System Error routines.

Changing the Compiler Block Size Options:

For **KEY**, code:

```
KEY=newpassword
```

For **ECB**, **PSB** and **BSB**, code:

```
ECB=newsize
PSB=newsize
BSB=newsize
(NOTE: maximum newsize for BSB is 4096)
```

For **ASB**:

To add an entry, code:

```
ASB=(newname,newsize)
```

To change the size of an existing entry, code:

```
ASB=(oldname,newsize)
```

To delete an entry (name and size), code:

```
ASB=(oldname,0)
```

NOTE: No more than five (5) **ASB** entries may be active at one time.

Changes are accepted during compilation (**EXEC**) runs and during **PERM** runs. It should be clear that 'EXEC' changes are only in effect for the duration of the compilation, and **PERM** changes establish new permanent defaults. Typically, **PERM** runs are only conducted by a Compiler Support Group, due to 'write protection' provided for the compiler components. The **KEY** password may only be changed during a **PERM** run, since during an **EXEC** run it is used to restrict access to the block size options, and must, therefore, match the established value.

Logically, the implementation of this enhancement does not alter the compiler's method of enforcing the block size limits. What has changed is that previously hard-coded values are now dynamically assigned. A brief explanation of block size evaluation follows:

For Automatic storage, the compiler attempts to find an argument for the **ALASC** macro (from all those stored with the Automatic Storage Block (**ASB**) option), which is associated with the smallest size equal to or greater than the byte count of all fields declared in Automatic storage. Sizes specified in the **ASB** option should be 8 bytes less than the true sizes of the desired Automatic storage blocks, since the first 8 bytes in the blocks are for the TPF System usage. For example, a size of 1047 describes a block of 1055 bytes.

For the Entry Control Block (**ECB**) and Based Storage Block (**BSB**), the total byte counts of all fields declared in these storage classes are compared, respectively, against the limits established with the ECB and BSB options.

For Program (Constant) storage, the byte count of compiler-generated Assembler Language instructions is compared against the limit established with the **PSB** option. Such byte count is approximate because the compiler does not expand TPF macros and thus cannot take into account a macro expansion byte count. It is therefore possible for a program size limit that was below the maximum during the compilation, to still be exceeded during assembly of the generated Assembler Language instructions.

In all cases, total byte counts are depicted in compiler output listings. Exceeding block size limits in effect for a compilation causes appropriate Error messages to be issued, as follows:

SBT0155E - AUTOMATIC STORAGE SIZE LIMIT HAS BEEN EXCEEED.

SBT0157E - PROGRAM SIZE LIMIT HAS BEEN EXCEEDED.

SBT0159E - ENTRY BLOCK SIZE LIMIT HAS BEEN EXCEEDED.

SBT0161E - THE AREA POINTED TO BY THE POINTER EXCEEDS BASED STORAGE SIZE LIMIT.

NOTE: Because of the format flag in storage blocks, the maximum size allowed on a Based Storage Block (BSB) should be 4095.

CHAPTER 8: SABRETALK IN AN INTERACTIVE ENVIRONMENT

SPECIAL NOTE: This chapter may be of limited use, as most examples involve using TSO for editing SABRETALK source code.

The purpose of this section is to familiarize programmers with the facilities available to them in an INTERACTIVE environment under control of an INTERACTIVE environment monitor (IEM), such as TSO, in the OS system. The assumption is made that the programmer is familiar with the general operation of an IBM INTERACTIVE environment. For those who are not, the following IBM publications are recommended:

- IBM OS/VS2 PROGRAMMING LIBRARY: TSO (order number GC28-0629)
- IBM OS/VS2 TSO TERMINAL USER'S GUIDE (order number GC28-0645)
- IBM OS/VS2 TSO Command Language Reference (order number GC28-0646)
- IBM Virtual Machine Facility / 370: CMS User's Guide (order number GC20-1819)
- IBM Virtual Machine Facility / 370: Terminal User's Guide (order number GC20-1810)
- IBM Virtual Machine Facility / 370: Quick Guide for Users (order number GX20-1926)

Throughout this section, examples will have the following format:

- data typed by the user will appear in lower case
- SYSTEM REPLIES WILL APPEAR IN UPPER CASE

CREATING A PROGRAM

With an INTERACTIVE monitor, programs are created, modified and saved in the EDIT mode. The programmer initiates the EDIT mode by using the EDIT command in the command mode. The format using TSO is:

```
edit  name  sabr ( old ) ( noscan )
                 ( new ) ( scan   )
                 (     ) (        )
```

EDIT, name and SABR are required. SABR tells the monitor that the program being edited is a SABRETALK program. New is also required when the program does not exist. The SCAN and NOSCAN options involve the syntax checking of the program's statements. If it is desired that syntax errors be displayed after every statement, SCAN must be specified:

```
edit  name  sabr new scan
```

If the programmer does not want such interruptions while building his program, he can accept the NOSCAN default:

```
edit  name  sabr new
```

After the program has been built, notification of all syntax errors can be obtained by use of the SCAN subcommand of the EDIT mode.

After the EDIT command for a NEW program has been given, the INTERACTIVE monitor facilitates the building of the program by automatically supplying statement numbers before each statement. This is known as the INPUT phase of the EDIT mode. After the monitor supplies the line number, the cursor is positioned at what would correspond to card-column one on a coding form. The programmer must remember that all SABRETALK statements must start in or after column two. The spacebar must be depressed before starting a statement.

The INPUT PHASE is terminated:

1) Implicitly when the syntax checker detects an error. This occurs only when SCAN was specified.

2) Explicitly by entering a null (empty) line.

The INPUT phase can be re-initiated by typing the INPUT sub-command or by entering a null line. After all statements have been typed, the programmer should return to EDIT mode, save his data set and **END** the EDIT phase. He can then compile his program. He may then re-enter EDIT to correct any errors not previously detected. This cycle can continue until the program is ready to be executed.

The following simulation shows what program creation looks like in practice:

```
READY
edit myprog sabr new scan
INPUT
00010   myprog: PROC;
00020           DCL (a,b) BIN;
00030           a = b;
00040           END myprog;
00050
EDIT
save
SAVED
end
READY
```

SYNTAX CHECKING

This section describes the Syntax Checker. Syntax checking is the process by which the system determines whether or not the source language statements in a given program are constructed properly. It detects syntax errors that can be found by scanning one complete statement. Diagnostic messages are generated for any errors of syntax that are detected.

This section also describes HELP information available for SABRETALK Functional descriptions and syntax of selected statements are available in EDIT mode, through the use of the HELP subcommand.

The Syntax Checker

The syntax checker allows a programmer in an INTERACTIVE environment to make corrections as he is keying in his program at the terminal. If a programmer makes a syntax mistake while entering his program, the syntax checker sends an error message to his terminal informing him of the error. The programmer can then re-enter the statement in the correct format and continue writing his program.

A programmer may use a batch technique to place a newly written program into an installation's library. If this library is available to the INTERACTIVE monitor, the program may be retrieved and scanned in its entirety for syntax errors. The programmer can then correct invalid statements. The result, whether the program was developed in a batch or an INTERACTIVE environment, is a syntactically correct program that is ready to be compiled.

The compiler may then be invoked by keying in a command from the terminal. This compile could uncover additional programmer errors. These would be inter-statement type errors such as undefined labels, duplicate names, data type incompatibilities, etc. If such errors are encountered, the programmer can correct his source from the terminal and compile the program again. When a clean compile is received the program is ready for execution testing.

The syntax checker performs single-statement validation as opposed to inter-statement checking. Its purpose is to eliminate syntax errors from a program before it is submitted for compilation. It is limited to the verification of punctuation, spelling of keywords, placement of keywords and operand in statements, etc. Inter-statement logic and meaning are evaluated when the program is compiled.

In the EDIT mode, notification of syntax errors is controlled by the SCAN/NOSCAN options and by the SCAN sub-command.

Syntax Checking New Statements

The user can request that each line he enters from the terminal in INPUT mode be immediately scanned for syntax errors by specifying the SCAN option with the EDIT command or ON with the SCAN sub-command. Before the record is scanned it is put in the user's data set. If a syntax error is found in a record just entered by the user, and an error message is displayed, the system will switch from INPUT to EDIT mode so that corrections can be made. The user returns to INPUT mode by entering a null line or the INPUT command.

Syntax Checking Old Statements

The SCAN sub-command can be applied to existing statements by specifying the SCAN subcommand with no operands. The entire program will be scanned for syntax errors. The syntax checker will complete the scan then display all errors found during the scan. The line number and an error message will be included for programmer reference.

Most errors halt syntax checking of a statement, that is, if more than one syntax error exists in a statement, only the first one will be reported. However, some errors encountered in structured **DECLARE** statements do not terminate the syntax checker and multiple error messages for a single statement can be generated.

Structure Mode

The syntax checker under an INTERACTIVE environment has flexibility that is not found in most other high-level language syntax checkers. It allows the programmer to code a structure a line at a time and have his input checked logically against the data already entered for the structure. This is the only variation from the rule that every entry to the syntax checker must be a complete statement or statements.

In order to provide this flexibility certain rules have been established. The programmer should acquaint himself with the logic of structure mode and the rules that apply before attempting a line-by-line scan of a structure.

Initializing Structure Mode

The syntax checker builds internal lists that permit the structure to be checked a line at a time. Each line of input can be compared against the data already entered to see if the rules of dimension and level have been violated. The syntax checker determines that the programmer desires to build a structure when it receives a **DECLARE** statement followed by a comma instead of a semi-colon.

```
INPUT
00010     DCL 1 rec,
```

Once the checker is in structure mode, it will retain all of the internal lists built for the structure until the structure is completed or certain unrecoverable syntax errors are encountered.

Completing the Structure

The example given above initialized structure mode. The programmer can now continue his structure a line at a time.

```
00020           2 date BIN,
00030           2 address CHAR(35);
```

Input line 30 above closes out the structure with a semi-colon. At this point the syntax checker is no longer in structure mode.

Error Handling in Structure Mode

Appropriate messages will be issued if errors are detected within the structure. Errors uncovered in a structure fall into two categories:

1) Errors that do not affect the internal lists. These errors are recoverable.

2) Syntax or logic errors that negate the validity of the internal lists. These errors are not recoverable.

Recoverable Errors

An example of such an error is depicted in the following simulation:

```
INPUT
00010     DCL 1 rec,
00020           2 partno BIN(33),
SBT014   20   DECLARATION OF A BINARY FIELD MUST BE 15 OR 31.
EDIT
```

The terminal is now in EDIT mode, ready for correction. The programmer should correct the error:

```
c /33/31/
00020           2 partno BIN(31),
```

He should then enter a null line to get back into INPUT mode.

```
INPUT
00030
```

The syntax checker is now ready to accept more data for the structure.

NOTE: The corrected line should not be rescanned. The internal lists have been set up for this line already. A rescan would cause line to be placed in the lists again and an unintentional syntax error would result.

Unrecoverable Errors

Certain syntax and logic errors occur in structure mode that negate the validity of the internal lists. When these errors are encountered the syntax checker will delete the lists and send messages that will guide the programmer in corrective action:

```
INPUT
00010     DCL 1 rec (10),
00020           2 mypart BIN,
00030           2 partlst,
00040             3 logs (6) BIN,
SBT029   40   DIMENSION HAS BEEN DECLARED WHEN ALREADY IN FORCE.
SBTCCC   CORRECT ERROR AND RESCAN FROM LINE 10.
EDIT
```

At this point the internal lists for the structure have been deleted and the syntax checker is effectively out of structure mode. The error should be corrected, and the entire structure rescanned:

```
c /(6) //
00040             3 logs BIN,
SCAN 10 40
```

The syntax checker will thus be forced back into structure mode, this time with the correct internal lists. The programmer can then go back into INPUT mode and complete the structure.

Rules for Structure Mode

1) Each line must end in a comma. Structure mode is terminated by a semi-colon.

2) Do not rescan the structure unless the syntax checker sends the request message:

CORRECT ERROR AND RESCAN FROM LINE nnnn.

3) Termination of structure mode can always be forced by entering a semi-colon.

Coding Standards

This topic is included to point out to the SABRETALK user in an INTERACTIVE environment, some coding standards for the syntax checker. The way a statement is entered under control of the INTERACTIVE monitor makes little difference to the compiler. However, the syntax checker works on a statement-by-statement basis. These tips are provided to assist the programmer.

The treatment of statements:

The syntax checker works on a statement-by-statement basis. It is therefore most efficient to enter one statement at a time. When the syntax checker is active during the INPUT mode each line entered must be a complete statement or an error will be issued. There are two types of statements recognized by the syntax checker. The first is a COMMENT statement. It begins with the composite /* and ends with */. The syntax checker is unable to detect a COMMENT that extends over one line. When a programmer wishes to enter a multi-line COMMENT he should begin and end each line with /* and */, respectively. The second type of statement is a Program statement. Every Program statement ends with a semi-colon. It may follow a COMMENT or another Program statement. An entire structure is considered a program statement just as any other type of statement except a COMMENT.

Unacceptable Statements

The following statements cannot be syntax checked without causing confusion to the Syntax Checker:

1) Statements containing non-syntax macros.

2) Non-SABRETALK Statements:

NOTE: Assembler Language statements, with or without labels, and the OPTIONS statement do not cause an error. The syntax checker disregards them.

Programmer Declared Functions

One assumption that the syntax checker makes is that all identifiers have been or will be declared within the program being entered. One exception is a programmer declared function. Two rules should be remembered:

1) Before a programmer defined function can be referenced, it must be declared.

2) If a change is made to an old program on a line with a programmer declared function, or a new line is to be inserted which contains reference to a programmer declared function, the line with the **DECLARE** for the FUNCTION must be scanned.

CHAPTER 9: SABRETALK COMPILER MESSAGES

The compiler issues information, warning, and error messages. The mutually exclusive options **MLEVEL0/MLEVEL1/MLEVEL2** determine which class or classes of messages will be printed at the end of the compilation.

There are seven types of messages:

A) Syntax errors

B) Programmer errors

C) Terminal errors

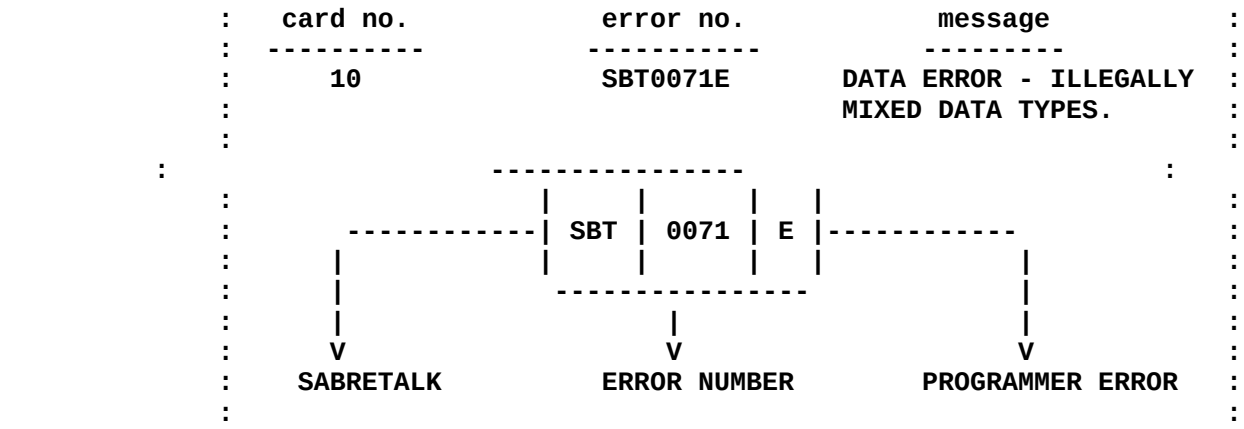
D) Internal Compiler errors

E) System (Loader) errors

F) Warning messages

G) Information messages

The type of message is encoded in the message printout:



The codes for errors and messages are:

1) S - Syntax error

2) E - Programmer error

3) T - Terminal error

4) C - Internal Compiler error

5) W - Warning message

6) I - Information message

7) L - Loader message

Source statements that generate errors are flagged in the source listing:

```
:   CARD NO.                SOURCE STATEMENT          :
:  ----------              ---------------------         :
:      20                   bbb = INDEX (aaa, hh, bbb);  :
:  ****ERROR****                                         :
:                                                        :
:      21                   bbb = bstr(b(i));            :
:  ****ERROR --------------->                            :
:                                                        :
```

The error encountered on card number 21 above is a Syntax error. (The grammatical rules of the language have been violated.) The 'arrow point' highlights the symbol causing the error.

The error messages themselves are listed at the end of the compilation. The messages associated with the examples above would be:

```
:   card no.       error no.       message                    :
:   --------       ---------       -----------------------    :
:   20             SBT0076E        THE ARGUMENT TYPE IS       :
:                                  INCORRECT.                 :
:                                                             :
:   21             SBT0002S        SYNTAX ERROR - THE LEFT    :
:                                  PARENTHESIS IS INVALID     :
:                                  AFTER THE NAME IN          :
:                                  CARD COLUMN 16.            :
:                                                             :
```

The Compiler also generates a message indicating the approximate byte count of the Assembler Language program associated with the source program. The figure provided at the bottom of the source listing excludes macro expansions. It includes all literals and constants before any optimization takes place. The number is provided to give the programmer some insight as to the approximate size of the program. If the programmer needs to know the exact size he must, of course, assemble the generated code.

Following are lists of messages by types. The message text itself will appear in upper case letters and may be followed by a short note (in upper and lower case) which provides a further explanation and/or advice.

Note: For Loader errors see the Sabretalk Installation Guide.

SEVERE PROGRAMMER ERROR MESSAGES

SBT0004E **ILLEGAL CHARACTER / STRING**
The statement contains a character used in a way that violates the language rules. Check the last character-string to see if it was closed properly.

SBT0005E **NESTED INCLUDES ARE ILLEGAL**
One may not use the **%INCLUDE** statement to read in a file that contains another **%INCLUDE** statement.

SBT0006E **FORMAT ERROR IN %INCLUDE STATEMENT**

SBT0008E **%INCLUDE FILE NOT FOUND**
The file name following the **%INCLUDE** has not been found on the library. Check spelling or contents of **%INCLUDE** library.

SBT0010E **%INCLUDEAF FILE NOT FOUND**
Same as SBT0008E, but **%INCLUDEAF** file is involved.

SBT0011E **_____ IS AN UNDEFINED SYMBOL.**
The identifier specified in the error message has not been declared in this program.

SBT0013E **CHARACTER SIZE IS NOT BETWEEN 1 AND 256.**
Length specification of **CHARACTER** field must be 1 - 256.

SBT0014E **DECLARATION FOR A BINARY FIELD MUST BE 15 OR 31.**
Length specification of **BINARY** field must be 15 or 31.

SBT0015E **DECIMAL POINT IS NOT BETWEEN 1 AND 15.**
15 is the maximum number of decimal digits that can be declared.

SBT0016E **THE DECIMAL POINT IS GREATER THAN THE SIZE OF THE FIELD.**
The first number specified in the length and precision of a **DECIMAL** number includes those digits to the left and right of the decimal point, the second number indicates the number of digits to the right of the decimal point only.

SBT0017E **BIT SIZE IS NOT BETWEEN 1 AND 32.**
Length specification of **BIT** field must be 1 - 32.

SBT0018E **THE DIMENSION ATTRIBUTE IS GREATER THAN 255 OR LESS THAN 1.**
Arrays have a maximum of 255 elements.

SBT0019E **THERE IS A CONFLICT IN ELEMENTARY ATTRIBUTES.**
More than one data type has been used to describe a single identifier.

SBT0020E **THE BASE IS NOT IN AUTO STORAGE OR IS NOT A POINTER.**
The implicitly declared pointer following the **BASED** attribute has previously been declared but not as a pointer in AUTOMATIC storage.

SBT0021E **THERE IS A CONFLICT IN STORAGE CLASSES.**
A storage class has previously been designated for this structure or element variable. The entire structure must reside in one class storage.

SBT0022E **_____ PREVIOUSLY DEFINED NAME.**
The identifier specified in the error message has previously been declared in this program. All identifiers must be unique.

SBT0023E **LEVEL1 HAS NOT BEEN DEFINED.**
The major structure of a structure must be defined as level 1:

```
DCL 1 able,
      2 baker CHAR(2),
        etc.
```

SBT0024E **LEVEL ERROR OR THE 'DEFINED' IS NOT AT THE END OF THE DECLARE.**
The major structure has been declared as other than level 1 or a variable has been declared with a level other than 1.

SBT0025E **THE END OF THE DECLARE IS NOT ELEMENTARY.**
A **DECLARE** statement has not been completed. Usually the error occurs when a level umber has been omitted from a structure making statement look like a multiple **DECLARE**:

```
     DCL 1 structure,
           strut2 CHAR(20);
```

SBT0026E **THERE IS A CONFLICT IN THE FACTORED LEVEL NUMBERS.**
When factoring attributes all identifiers must have the same level number. Such an error would be:

```
DCL  1  rec,
        2  (fld,flt),
           3  (col,crd)     BIN;
```

SBT0027E **THE OBJECT OFTHE 'DEFINED' HAS NOT BEEN PREVIOUSLY DECLARED.**
The **DEFINED** attribute must be based on an identifier declared in this program.

SBT0028E **THE STORAGE CLASS HAS BEEN DECLARED AT A LEVEL GREATER THAN 1.**
The storage class of a **BASED** structure must be declared at level 1. It may not appear at any other level:

```
DCL 1 rec BASED (iptr),
      2 head CHAR(8),
        etc.
```

SBT0029E **DIMENSION HAS BEEN DECLARED WHEN ALREADY IN FORCE.**
An array cannot contain another array. The following **DECLARE** statement would create this error:

```
DCL 1 record (10),
      2 data (5) BIN;
```

SBT0030E **THERE ARE 2 'DEFINES' FOR THIS ITEM.**
The item declared with the **DEFINED** attribute is based on an item that has also been declared with the **DEFINED** attribute.

SBT0031E **THE LEVEL IS EITHER GREATER THAN 256 OR LESS THAN 1.**
Level, if specified, must be between 1 and 256. If not specified, (as in a single variable,) it is assumed to be 1.

SBT0032E **SYSEQ REFERENCE CONTAINS IMBEDDED BLANK(S).**

SBT0033E **FUNCTION TABLE EXCEEDED.**
There are more than 32 programmer declared functions in one program segment (external procedure.)

SBT0034E **BASE TABLE EXCEEDED. MORE THAN 92 POINTERS WERE DECLARED.**

SBT0035E **LENGTH OF RECEIVING FIELD IS SHORTER THAN SENDING FIELD, MVCL DOES NOT TAKE PLACE.**

SBT0036E **ERROR - THE SEMICOLON AFTER THE THEN/ELSE IS INVALID.**

SBT0037E **ERROR - THE SEMICOLON AFTER THE SEMICOLON IS INVALID.**

SBT0038E **THERE IS AN UNDEFINED SYMBOL IN THIS STATEMENT.**
The Compiler was unable to determine what symbol in the statement was the one not declared in the program. Check identifiers in the statement against **DECLARE** statements.

SBT0039E **DECLARATION FOR A DEC FLOAT FIELD MUST BE 6 OR 16.**

SBT0040E **THERE IS A CONFLICT BETWEEN DEFINED AND BASED ATTRIBUTES FOR THIS ITEM.**

SBT0042E **EMBEDDED BAL STATEMENTS NOT ALLOWED.**

SBT0044E **NAME IN ALLOCATE STATEMENT IS UNDEFINED.**

SBT0045E **STORAGE CLASS FOR NAME IN ALLOCATED STMNT ISNT BASED OR RENTED**

SBT0046E **NAME IN FREE STATEMENT IS UNDEFINED.**

SBT0047E **STORAGE CLASS FOR NAME IN FREE STMNT IS NOT BASED OR RENTED.**

SBT0048E **RETURN STATEMENT ILLEGAL IN MAIN PROCEDURE.**
Main (external) **PROCEDURE** may be entered and exited by the use of TPF ENTER and BACK type macros only.

SBT0049E **CONSTANT IS ILLEGAL AS A PARAMETER FOR AN INTERNAL PROCEDURE.**

SBT0050E **LABEL ON STATEMENT HAS BEEN DEFINED (NOT AS A FUNCTION).**
The label on the **PROCEDURE** statement has been declared as an identifier in this program.

SBT0051E **MAIN PROCEDURE CANNOT BE A FUNCTION.**

SBT0052E **LABEL ON FUNCTION HAS BEEN FOUND ON PREVIOUS PROC STATEMENT.**
All labels in a program must be unique.

SBT0053E **TOO MANY INTERNAL PROCEDURES AND PROGRAMMER DEFINED FUNCTIONS.**
A maximum of 50 internal procedures and/or programmer declared functions are allowed in a given program.

SBT0054E **LENGTH OF CSTR (CHARACTER-STRING) IS GREATER THAN 256.**
A character-string must be from 1 to 256 characters in length. This restriction applies to the length specified in the **CSTR** function.

SBT0055E **LENGTH OF BSTR (BIT-STRING) IS GREATER THAN 32.**
A **BIT**-string must be from 1 to 32 bits in length. This restriction applies to the length specified in the **BSTR** function.

SBT0056E **LENGTH OF NSTR (NUMERIC STRING) IS GREATER THAN 15.**
Because **NSTR** data is presumed to be data for arithmetic operations, its size is limited to the 15 digit restrictions of **DECIMAL** arithmetic.

SBT0057E **2ND PARAM OF CSTR, BSTR OR NSTR CANT BE AN EXPRESSION WITHOUT 3RD.**

SBT0058E **2ND PARAM OF CSTR, BSTR OR NSTR CANT BE A VARIABLE WITHOUT 3RD PARAM.**

SBT0059E **EXPRESSION ILLEGAL AS 2ND PARAMETER OF BSTR (BIT-STRING).**
The second parameter of the **BSTR** function must always be a constant if it is specified.

SBT0060E **THE FIRST ARGUMENT OF A BSTM FUNCTION IS ILLEGAL.**
The first parameter of **BSTM** must be a Variable or an expression.

SBT0061E **THE SECOND ARGUMENT OF A BSTM FUNCTION IS ILLEGAL.**
The second parameter of **BSTM** must be a **BINARY** literal.

SBT0062E **NO ARGUMENT OR ILLEGAL ARGUMENT SPECIFIED ON MACRO STATEMENT.**
Check TPF macro manual and check macros supported by SABRETALK

SBT0063E **NO LITERAL OR ILLEGAL LITERAL SPECIFIED FOR MACRO STATEMENT.**
Check TPF macro manual and check macros supported by SABRETALK

SBT0064E **STORAGE CLASS IS NOT PROGRAM BASED OR CONSTANT BASED.**
The identifier specified in the **CONST** statement has not been defined in **CONSTANT** storage.

SBT0065E **ILLEGAL MACRO NAME SPECIFIED IN MACRO STATEMENT.**
More than one macro has been used in a statement or a macro has been used as an identifier.

SBT0066E **ILLEGAL NESTING OF CSTR, BSTR, OR NSTR FUNCTION.**
The string functions may not contain other string functions as parameters.

SBT0067E **INVALID REGISTER NOTED.**
The register number employed in a register assignment is greater than 15.

SBT0068E **_____ IS AN UNDEFINED LABEL.**
The label designated in the error message has not been defined in the program as a statement label or label variable.

SBT0069E **_____ IS AN INVALID FUNCTION.**
The function designated in the error message either has not been declared as a function, or has a discrepancy in parameters. The number of parameters passed in the function invocation does not match that on the **PROCEDURE** statement.

SBT0070E **A BIT STRING IS NOT DIVISIBLE BY 8.**

SBT0071E **DATA ERROR - ILLEGALLY MIXED DATA TYPES.**
The statement flagged with the error contains an operation that is illegal for the two types of data involved. The operation could be assignment, arithmetic, relational, logical or string.

SBT0072E **CONTINUATION CARDS FOR THIS MACRO MUST BEGIN PRIOR TO CARD COL. 17.**

SBT0073E **THE FUNCTION NAME IS NOT IN THE LIST OF LEGAL FUNCTION NAMES.**
The function name is not a valid built-in function and has not been defined as a programmer declared function.

SBT0074E **THERE ARE AN INCORRECT NUMBER OF FUNCTION ARGUMENTS.**
The number of parameters being passed to a built-in function does not match the required input.

SBT0075E **ARGUMENT WITHIN BUILT-IN FUNCTION INCOMPLETE.**
Max built-in function requires at least two arguments within parentheses, and/or parentheses must be balanced.

SBT0076E **THE ARGUMENT IS ILLEGAL FOR THE BUILT-IN FUNCTION.**
The parameter passed to the built-in function is not legal for that function.

SBT0077E **AN INDEX FUNCTION HAS AN INVALID SIZE.**
The first parameter of the **INDEX** built-in function must be a character-string larger than the second.

SBT0078E **THE FIRST ARGUMENT OF A LSTR FUNCTION CANNOT BE A BIT STRING CHARACTER.**

SBT0079E **BASE IS NOT IN AUTOMATIC STORAGE NOR IS IT A POINTER.**
The error is normally a result of incorrectly using a pointer qualifier.

SBT0080E **THE FIRST ARGUMENT OF A LSTR FUNCTION CANNOT BE A LITERAL.**

SBT0081E **LITERAL ERROR -- ILLEGAL OR UNDEFINED LITERAL TYPE.**

SBT0082E **LITERAL ERROR -- BIT-STRING LENGTH IS GREATER THAN 32.**

SBT0083E **LITERAL ERROR -- CONTENTS OF THE BIT-STRING ARE NOT 0 OR 1.**

SBT0084E **LITERAL ERROR -- TOO LARGE FOR ARITHMETIC DATA TYPE.**

SBT0085E **LITERAL ERROR -- NO DECIMAL POINT FOUND IN CAD FIELD.**

SBT0086E **LITERAL ERROR -- HEX FIELD GREATER THAN A FULL WORD.**
Because the hexadecimal literal is another way of expressing a **BIT**-string literal, it is limited to a fullword (32 bits) of valid hexadecimal digits.

SBT0087E **LITERAL ERROR -- INVALID CHARACTER IN THE HEX LITERAL.**
The label on the **END** statement does not match the label on the **DO** group in question.

SBT0088E **THE SECOND ARGUMENT IS INVALID FOR AN ELEMENT OF A SUBSCRIPT ENTRY**

SBT0089E **DO LOOP ERROR. THE LABEL ON THE END STATEMENT IS UNDEFINED.**

SBT0091E **ATTRIBUTES IN FACTOR LIST.**
Identifiers may be factored only if all the attributes in the **DECLARE** statement apply to all the identifiers.

SBT0092E **ERROR - MISSING OR MISPLACED END STATEMENT(S).**

SBT0093E **EXTRANEOUS END STATEMENT.**

SBT0096E **THE FIELD BEING SUBSCRIPTED IS UNDEFINED.**

SBT0097E **INDEX ERROR. A NON-DIMENSIONED ITEM.**
The identifier being subscripted has not been declared as an array.

SBT0099E **INDEX ERROR. THE DATA TYPE FOR THE INDEX IS INVALID.**
The subscript variable may be **BIT, BIN, DEC** or **NCS.**

SBT0100E **DO LOOP ERROR. WHILE OPERAND IS NOT A PROGRAMMER DEFINED FUNCTION.**

SBT0101E **ILLEGAL EXPRESSION IN INDEX.**
The expression is illegal for the **INDEX** built-in function.

SBT0103E **INDEX ERROR. MORE THAN ONE VARIABLE IN INDEX.**
Only one variable is permitted in a subscript expression.

SBT0104E **INDEX ERROR. NEGATIVE VARIABLE.**
A prefix minus is not allowed in a subscript expression.

SBT0108E **INVALID DATA TYPE FOR THE SIGN FUNCTION.**
The **SIGN** built-in function may only be used on arithmetic data types. No string data is allowed.

SBT0111E **INVALID REGISTER BEING USED AS A PARAMETER ON A MACRO.**
The register number assigned in a macro statement conflicts with standard system register assignments. Check a TPF macro manual.

SBT0113E **REGISTER BEING STORED HAS BEEN ALTERED PRIOR TO THE START MACRO.**
The **START** macro should be the first executable statement. If it is not, registers being stored by **START** may have been loaded prior to the stores.

SBT0121E **INCORRECT DCL, INCOMPLETE INFO - CHECK ITEMS IN THIS STATEMENT**
The error will normally point to an executable statement. However, the cause is usually a **DECLARE** within a structure referenced within an expression. Check structures for improper declares.

SBT0122E **_____ IS A POINTER THAT IS BEING LOADED BUT HAS NOT BEEN INITIALIZED.**
A register is being loaded with an uninitialized pointer.

SBT0124E **CONSTANT PREVIOUSLY INITIALIZED**
The constant referenced in the **CONST** statement has already been initialized.

SBT0125E **ILLEGAL USE OF LOGICALS IN AN IF STATEMENT. USE PARENTHESES.**

SBT0126E **ILLEGAL USE OF PARENTHESIS.**

SBT0127E **SIZE OF CONSTANT DOES NOT AGREE WITH DECLARED SIZE**
The constant has been incorrectly initialized.

SBT0128E **THIRD PARAMETER OF VSTR FUNCTION MAY NOT BE CHARACTER STRING**

SBT0136E **ECS FEATURE HAS NOT BEEN INSTALLED IN THIS VERSION.**

SBT0150E **THE NUMBER OF DECIMAL DIGITS AFTER THE DECIMAL POINT > 15.**
The maximum number of digits in **DEC** attribute is 15.

SBT0155E **AUTOMATIC STORAGE SIZE LIMIT HAS BEEN EXCEEDED.**

SBT0156E **PICTURE SPECIFIES MORE THAN 15 DIGITS AND 32 CHARACTERS.**

SBT0157E **PROGRAM SIZE LIMIT HAS BEEN EXCEEDED.**

SBT0158E **_____ IS A DECLARED CONSTANT WHICH NEEDS TO BE INITIALIZED.**

SBT0159E **ENTRY BLOCK SIZE LIMIT HAS BEEN EXCEEDED.**

SBT0160E **PICTURE SPECIFIES MORE THAN 15 DIGITS.**

SBT0162E **CHECK ALL MACROS - ONLY 256 CHARS ARE PASSED TO ASSEMBLER.**

SBT0163E **PICTURE ENDS WITH PARENTHESIS.**
PICTURE specification should end with a single quote.

SBT0164E **PICTURE SPECIFIES MORE THAN 32 CHARS.**
The maximum **PICTURE** size for Numeric character-string fields is 32 characters only 15 of which may specify digit positions.

SBT0165E **NO NUMBER BETWEEN LEFT AND RIGHT PARENTHESES.**

SBT0167E **ILLEGAL 9 ENTRY.**
Check placement of '**9**' characters.

SBT0168E **ILLEGAL V ENTRY.**
Implied decimal point used with character-string is illegal.

SBT0169E **MORE THAN ONE IMPLIED POINT.**
Only one implied decimal point is allowed.

SBT0170E **ILLEGAL R ENTRY.**

SBT0171E **LEADING AND TRAILING SIGN.**
Only one sign is permitted.

SBT0173E **ILLEGAL SIGN ENTRY.**
Not one of: **S + -**

SBT0175E **ILLEGAL C ENTRY.**
Check **CR** placement**.**

SBT0176E **CR/DB MAY NOT BE CODED IF A SIGN CHARACTER IS ALSO CODED.**

SBT0177E **ILLEGAL D ENTRY.**
Check **DB** placement**.**

SBT0178E **ILLEGAL INSERTION CHARACTER ENTRY.**
Not one of: **. , / B**

SBT0179E **ILLEGAL DOLLAR SIGN OR Z ENTRY.**
Check placement of **$** or **Z**.

SBT0180E **INVALID END.**
No single quote.

SBT0181E **INVALID CHARACTER.**
Not a valid **PICTURE** character.

SBT0183E **ILLEGAL LEADING CHARACTER.**
Not one of: **9 $ - + Z ***

SBT0184E **ILLEGAL FLOATING CHARACTER.**
Not one of: **$ S + - ***

SBT0186E **STATEMENT NOT WITHIN COLUMNS 2 - 71.**

SBT0187E **REQUIRED ERROR EXIT IN MACRO CALL HAS NOT BEEN SPECIFIED.**

SBT0188E **STATEMENT APPEARS TOO LONG, CHECK SEMI COLONS AND QUOTES.**
The Compiler encountered an undetermined error situation. Usually caused by a previous incomplete statement.

SBT0189E **INVALID EDIT PATTERN FOR DECIMAL FLOAT DATA TYPE -EDIT PATTERN MUST HAVE AN E.**

SBT0190E **INVALID EDIT PATTERN FOR DECIMAL DATA TYPE - EDIT PATTERN CAN NOT HAVE AN E.**

SBT0191E **INVALID EDIT PATTERN FOR FLOAT DATA TYPE - PATTERN CONTAINING A V MUST HAVE A DECIMAL POINT.**

SBT0192E **INVALID EDIT PATTERN FOR FLOAT DATA TYPE - PATTERN CONTAINING A DECIMAL POINT MUST HAVE A V .**

SBT0193E **_____ A REQUIRED MACRO ERROR EXIT IS ILLEGAL (PROCEDURE).**

SBT0243E **VARIABLES WITHIN A CSTR OR NSTR MAY NOT BE SUBSCRIPTED.**
Calculate the value in a separate statement.

SBT0244E **EXPRESSIONS CONTAINING THE CONST 0 MAY NOT BE SUBSCRIPTED.**
The default for a subscript variable is 1.

SBT0245E **_____ DECLARED AS A FUNCTION HAS NOT BEEN INCLUDED IN THIS PROGRAM.**
The label was declared as a function. The function itself was not coded.

SBT0246E **A VBL BEING LOADED INTO A REG IS > 4 OR ITS TYPE IS INVALID.**
Redefine the variable.

SBT0247E **A VBL BEING STORED FROM A REG IS > 4 OR ITS TYPE IS INVALID.**
Redefine the variable.

SBT0248E **THE FUNCTION IS INCOMPATIBLE WITH THE CONCATENATE OPERATION.**
Perform the concatenation in a separate statement.

SBT0249E **ERROR EXIT ADDRESS IN MACRO HAS NOT BEEN DEFINED.**
Insure error exit routine has a label.

SBT0250E **ILLEGAL OP. RECEIVING FIELD WITHIN THE PROGRAMS BASE.**
An attempt was made to modify program storage.

SBT0251E **INDEX BEYOND INITIAL FIELD, NO 3RD PARAMETER OR FIELD TOO LARGE.**

SBT0252E **TOO MANY ITEMS OF 1 DATA TYPE ON LEFT OF ASSIGNMENT STATEMENT.**
A maximum of 32 multiple assignments to fields of the same data type are allowed in a single statement.

SBT0253E **ENTRC MACRO - REGISTER WAS LOADED MORE THAN ONCE.**

SBT0254E **ENTRC MACRO - ECS CANNOT BE PASSED AS A PARAMETER.**

SBT0255E **ENTRC - ALL REGS ARE PARAMS, BASED DATA CAN'T BE LOADED OR STORED.**
There is no register available for use as a pointer to **BASED** storage. Move the data from **BASED** to **AUTO** storage for purposes of parameter passing.

SBT0256E **MACRO ARGUMENT LIMIT OF 20 HAS BEEN EXCEEDED.**

SBT0257E **RETURN STATEMENT MISSING FROM PROCEDURE**

SBT0258E **A PROCEDURE MUST BE CALLED. A GO TO <PROC> IS INVALID.**

TERMINAL ERROR MESSAGES

SBT0003T **OVERFLOW OF STACK SPACE, STATEMENT TOO LONG**
Reduce length of statement.

SBT0007T **OVERFLOW OF %INCLUDEAF STACK, TOO MANY %INCLUDEAF STATEMENTS**
Eliminate some of the **%INCLUDEAF** statements.

SBT0009T **OVERFLOW OF NAME STACK, TOO MANY IDENTS IN PROGRAM**
The program being compiled contains too many identifiers. It probably is much larger than the TPF system will handle, as well.

SBT0032T **ERROR TABLE OVERFLOW DUE TO TOO MANY PROGRAMMER ERRORS. COMPILATION IS TERMINATED.**
Correct the errors that have been specified and re-submit the program for compilation.

SBT0070T **TERMINAL ERROR. FIRST ACCESS MUST BE P1PSEDR. INTERNAL COMPILER ERROR.**

SBT0078T **TERMINAL ERROR. PSEUDO REGISTER SAVE AREA TABLE SIZE EXCEEDED. THE STATEMENT CONTAINS TOO MANY OPERATIONS.**

SBT0088T **TERMINAL ERROR. NO DO INFORMATION TABLE AVAILABLE TO PROCESS DO LOOPS. CARD NO. MAY BE INVALID.**

SBT0090T **TERMINAL ERROR. DO LOOP ERROR. THERE IS NO DO HEADER. CARD NO. MAY BE INVALID.**

SBT0122T **LITERAL TABLE OVERFLOW. TOO MANY LITERALS AND/OR CONSTANTS IN PROGRAM.**
Reduce the size of the program and eliminate some literals.

SBT0147T **TERMINAL ERROR. OUTPUT BUFFER LIMITS ARE EXCEEDED. COMPILATION IS TERMINATED.**
Reduce the size of the program.

SBT0186T **INSUFFICIENT CORE ALLOCATED FOR THIS COMPILATION. RUN ABORTED.**
Reduce size of program or use **%INCLUDEAF** for data declarations.

SBT0188T **TERMINAL ERROR. ILLEGAL STATEMENT CAUSED INTERNAL COMPILER OVERFLOW**
Remove the statement.

SBT0240T **ERROR ENCOUNTERED IN COMPILATION OF %INCLUDEAF. %INCLUDEAF FILE NOT UPDATED.**

SBT0241T **AUTOMATIC STORAGE DECLARES NOT PERMITTED IN %INCLUDEAF FILE. %INCLUDEAF FILE NOT UPDATED.**
%INCLUDEAF must be **BASED** or **ENTRYBLOCK** storage class.

SBT0242T **RENTED STORAGE DECLARES NOT PERMITTED IN INCLUDEAF FILE. INCLUDEAF FILE NOT UPDATED.**

SBT0256T **NO EXECUTABLE STATEMENTS HAVE BEEN GENERATED AS A RESULT OF THIS COMPILATION.**

SBT0257T MULTIPLE INCLUDEAF FILE HAD ILLEGAL DEFINES. INCLUDEAF FILE NOT UPDATED

SBT0264T DO INFORMATION TABLE CAPACITY EXCEEDED DUE TO AN EXCESSIVE NUMBER OF DO LOOPS IN PROGRAM.

INTERNAL COMPILER ERRORS

Internal Compiler error messages, those ending in the letter C, should not concern the SABRETALK programmer. If a statement causes an Internal Compiler error it should be reported to the SABRETALK GROUP. Because these errors do not directly relate to the SABRETALK programmer, they have not been included in this Guide.

WARNING MESSAGES

SBT0012W WARNING - IS AN UN-INITIALIZED POINTER.

SBT0013W WARNING - INTERPRETATION OF THIS STATEMENT IS OBSCURE.

SBT0014W WARNING - CONSTANT DECLARED AS BIN(15) HAS BEEN ALLOCATED AS BIN(31).

SBT0032W WARNING - SYSEQ REFERENCE CONTAINS IMBEDDED BLANK(S).

SBT0034W WARNING - MACRO PARAMETER CONTAINS IMBEDDED BLANK(S).

SBT0037W WARNING - MACRO PARAMETER REFERENCES A PROCEDURE NOT TAG. BRANCH MAY CAUSE PROBLEM.

SBT0049W ARGUMENTS ON MAIN PROCEDURE STATEMENT WILL BE IGNORED.

SBT0061W FIRST SOURCE STATEMENT IS NOT A PROCEDURE STATEMENT.

SBT0129W WARNING - SEMICOLON MAY BE INVALID AFTER THE ELSE.
Verify that the semicolon is correctly placed in the statement after the **THEN**.

SBT0130W WARNING - MISSING OR MISPLACED END STATEMENT(S) SIMULATED BY END OF INPUT.

SBT0136W WARNING - TRUNCATION MAY OCCUR - RECEIVING FIELD IS SHORTER THAN SENDING FIELD.

SBT0155W WARNING - EXTERNAL PROCEDURE END STATEMENT IS EXTRA OR MISPLACED - END STATEMENT IS IGNORED.

SBT0156W WARNING - LABEL ON EXTERNAL PROCEDURE END STATEMENT DOES NOT MATCH NAME OF MAIN PROCEDURE.

SBT0157W WARNING - LAST SOURCE STATEMENT IS NOT AN END STATEMENT / END MAY BE MISSING OR MISPLACED.

SBT0158W WARNING - END STATEMENT IS EXTRA OR MISPLACED - END STATEMENT IS IGNORED.

SBT0160W **WARNING - IMPROPER USE OF BUILT-IN FUNCTIONS WITHIN AN TPF MACRO AND/OR ---(NOTE: THIS WARNING ISSUED AS A PREFIX TO A RELATED SYNTAX MESSAGE).**

SBT0161W **WARNING - AREA POINTED TO BY THE POINTER EXCEEDS BASED-STORAGE LIMITS.**

SBT0166W **WARNING - MISSING OR MISPLACED END STATEMENT(S) SIMULATED BY EXTERNAL PROCEDURE END STATEMENT.**

SBT0167W **WARNING - MISSING OR MISPLACED END STATEMENT(S) PRIOR TO A RETURN IN AN INTERNAL PROCEDURE.**

SBT0172W **WARNING - LABEL ON END STATEMENT IS INCORRECTLY USED - LABEL IS IGNORED.**

SBT0173W **WARNING -- ARRAY ELEMENT IS NOT INDEXED, DEFAULTS TO ONE.**

SBT0174W **WARNING -- ARRAY ELEMENT INDEX IS NOT SUPPORTED BY CASE. INDEX DEFAULTS TO ONE.**

SBT0180W **WARNING -- ROUND FUNCTION - PLACE VALUE BEYOND RANGE OF TARGET. NO ROUND WILL OCCUR.**

SBT0190W **WARNING -- LENGTH OF EDIT PATTERN EXCEEDS THE PRECISION OF THE NUMBER BEING EDITED.**

SBT0191W **WARNING -- EDIT PATTERN CONTAINS A V BUT DOES NOT CONTAIN A DECIMAL POINT.**

SBT0192W **WARNING -- EDIT PATTERN CONTAINS A DECIMAL POINT BUT DOES NOT CONTAIN A V.**

SBT0193W **WARNING -- THIS IS A RECURSIVE CALL -- MAY RESULT IN A LOOP.**

INFORMATION MESSAGES

SBT0500I **OPTIMIZATION ERROR FROM POFORM - INFORM SABRETALK GROUP.**

SYNTAX CHECKER MESSAGES

SBT001S **DECIMAL NUMBER > 15.**
A **DECIMAL** field can contain a maximum of fifteen digits.

SBT002S **BIT SIZE > 32.**
A **BIT**-string can only be defined with a length of 1 to 32.

SBT003S **INVALID HEX CHARACTER.**
A character in a hexadecimal literal was not 0 through 9 or A through F.

SBT004S **ILLEGAL CHAR OR STRING > 256.**
Check the character-string length and make certain that there is no unpaired single quote.

SBT005S **SYNTAX ERROR IN %INCLUDE OR %INCLUDEAF.**

SBT006S **INCOMPLETE STATEMENT.**
The syntax checker must have an entire statement, enter the rest of the statement, or a semi-colon.

SBT007S **STATEMENT CONTAINS UNCLOSED COMMENT.**
To continue comment, enter /* at the start of the next line.

SBT009S **ILLEGAL BIT CHARACTER.**
A character in a **BIT**-string literal was not 0 or 1.

SBT010S **DANGLING STRUCTURE.**
A structure was not completed. Finish or enter a semi-colon.

SBT013S **CHARACTER SIZE IS NOT BETWEEN 1 AND 256.**

SBT014S **DECLARATION FOR A BINARY FIELD MUST BE 15 OR 31.**
Length of a **BINARY** field must be 15 or 31, plus the sign bit.

SBT015S **DECIMAL POINT IS NOT BETWEEN 1 AND 15.**
15 is the maximum number of **DECIMAL** digits that can be declared.

SBT016S **THE DECIMAL PART IS > THE SIZE OF THE FIELD.**
The second number indicates the decimal part - it must be smaller than the first number.

SBT017S **BIT SIZE IS NOT BETWEEN 1 AND 32.**

SBT018S **THE DIMENSION ATTRIBUTE IS > 255 OR < 1.**
Arrays have a maximum of 255 elements.

SBT019S **THERE IS A CONFLICT IN ELEMENTARY ATTRIBUTES.**
More than one data type has been used to describe a single identifier.

SBT021S **THERE IS A CONFLICT IN STORAGE CLASSES.**
The entire structure must reside in one storage class.

SBT023S **LEVEL 01 HAS NOT BEEN DEFINED.**
The major structure must be defined as a level 1.

SBT024S **LEVEL ERROR OR THE DEFINED IS NOT AT THE END OF THE DECLARE.**
The major structure has been declared as other than level 1 or an individual variable has been declared with a level other than 1.

SBT025S **THE END OF THE DECLARE IS NOT ELEMENTARY.**
A **DECLARE** statement has not been completed.

SBT026S **THERE IS A CONFLICT IN THE FACTORED LEVEL NUMBERS.**
When factoring attributes, all identifiers must have the same level number.

SBT028S **THE STORAGE CLASS HAS BEEN DECLARED AT A LEVEL > 1.**
The storage class of a **BASED** structure must be declared at level 1.

SBT029S **DIMENSION HAS BEEN DECLARED WHEN ALREADY IN FORCE.**
An array cannot contain another array.

SBT031S **THE LEVEL IS > 256 OR < 1.**

SBT091S **ATTRIBUTES IN FACTOR LIST.**
Identifiers may be factored only if all the attributes in the **DECLARE** statement apply to all the identifiers.

SBT163S **PICTURE ENDS WITH PARENTHESIS.**
PICTURE specification should end with a single quote.

SBT164S **PICTURE TOO LARGE.**
Maximum **PICTURE** size for a Numeric character-string is 15, for an Edited character-string is 32.

SBT165S **NUMBER BETWEEN LEFT AND RIGHT PARENS MISSING.**

SBT167S **ILLEGAL 9 ENTRY.**
Check placement of '**9**' characters.

SBT168S **ILLEGAL V ENTRY.**
Implied decimal point used with character-string is illegal.

SBT169S **MORE THAN ONE IMPLIED POINT.**
Only one implied decimal point is allowed.

SBT170S **ILLEGAL R ENTRY.**

SBT171S **LEADING AND TRAILING SIGN.**
Only one sign permitted.

SBT173S **ILLEGAL SIGN ENTRY.**
Not one of: **S + -**

SBT175S **ILLEGAL C ENTRY.**
Check **CR** placement.

SBT177S **ILLEGAL D ENTRY.**
Check **DB** placement.

SBT178S **ILLEGAL INSERTION CHARACTER ENTRY.**
Not one of: **. , / B**

SBT179S **ILLEGAL DOLLAR SIGN OR Z ENTRY.**
$ or **Z** in wrong place.

SBT180S **INVALID END.**
No single quote.

SBT181S **INVALID CHARACTER.**
Not a valid **PICTURE** character.

SBT184S **ILLEGAL FLOATING CHARACTER.**
Not one of: **$ S + - ***

SBT185S **UNCLOSED QUOTE.**
A string started with a quote has not been terminated.

SBT186S **STATEMENT NOT WITHIN COLUMNS 2 - 71.**

APPENDIX A:

The ECB shown here is an example of how one would describe a data record to be included in the INCLUDE library. This is provided for information only, and should be used only after it is compared to the installation requirements.

EXAMPLE FORMAT OF AN ENTRY CONTROL BLOCK

Format of the Entry Control Block: (an **INCLUDE** Member)

```
DECLARE 1 EB0EB ENTRYBLOCK,       /* 1055 BYTE ECB                */
          2 CE1CHW PTR,           /* CHAIN WORD                   */
          2 CE1BAD PTR,           /* POST INTERRUPT BRANCH ADDR   */
          2 CE1WKA,               /* WORK AREA                    */
            3 EBW000F,
              4 (EBW000,EBW001,EBW002,EBW003) CHAR (1),
            3 EBW004F,
              4 (EBW004,EBW005,EBW006,EBW007) CHAR (1),
            3 EBW008F,
              4 (EBW008,EBW009,EBW010,EBW011) CHAR (1),
            3 EBW012F,
              4 (EBW012,EBW013,EBW014,EBW015) CHAR (1),
            3 EBW016F,
              4 (EBW016,EBW017,EBW018,EBW019) CHAR (1),
            3 EBW020F,
              4 (EBW020,EBW021,EBW022,EBW023) CHAR (1),
            3 EBW024F,
              4 (EBW024,EBW025,EBW026,EBW027) CHAR (1),
            3 EBW028F,
              4 (EBW028,EBW029,EBW030,EBW031) CHAR (1),
            3 EBW032F,
              4 (EBW032,EBW033,EBW034,EBW035) CHAR (1),
            3 EBW036F,
              4 (EBW036,EBW037,EBW038,EBW039) CHAR (1),
            3 EBW040F,
              4 (EBW040,EBW041,EBW042,EBW043) CHAR (1),
            3 EBW044F,
              4 (EBW044,EBW045,EBW046,EBW047) CHAR (1),
            3 EBW048F,
              4 (EBW048,EBW049,EBW050,EBW051) CHAR (1),
            3 EBW052F,
              4 (EBW052,EBW053,EBW054,EBW055) CHAR (1),
            3 EBW056F,
              4 (EBW056,EBW057,EBW058,EBW059) CHAR (1),
            3 EBW060F,
              4 (EBW060,EBW061,EBW062,EBW063) CHAR (1),
            3 EBW064F,
              4 (EBW064,EBW065,EBW066,EBW067) CHAR (1),
            3 EBW068F,
              4 (EBW068,EBW069,EBW070,EBW071) CHAR (1),
            3 EBW072F,
              4 (EBW072,EBW073,EBW074,EBW075) CHAR (1),
            3 EBW076F,
              4 (EBW076,EBW077,EBW078,EBW079) CHAR (1),
            3 EBW080F,
              4 (EBW080,EBW081,EBW082,EBW083) CHAR (1),
            3 EBW084F,
```

```
    4 (EBW084,EBW085,EBW086,EBW087) CHAR (1),
  3 EBW088F,
    4 (EBW088,EBW089,EBW090,EBW091) CHAR (1),
  3 EBW092F,
    4 (EBW092,EBW093,EBW094,EBW095) CHAR (1),
  3 EBW096F,
    4 (EBW096,EBW097,EBW098,EBW099) CHAR (1),
  3 EBW100F,
    4 (EBW100,EBW101,EBW102,EBW103) CHAR (1),
  3 EBW104F,
    4 (EBSW01,EBSW02,EBSW03,EBRS01) BIT (8),
  3 EBW108F,
    4 (EBCM01,EBCM02,EBCM03,EBER01) BIT (8),
2 CE1FA0 BIT(32),                          /* FARW - 0  */
2  CE1FM0 BIT(32),
2 CE1FA1 BIT(32),                          /* FARW - 1  */
2  CE1FM1 BIT(32),
2 CE1FA2 BIT(32),                          /* FARW - 2  */
2  CE1FM2 BIT(32),
2 CE1FA3  BIT(32),                         /* FARW - 3  */
2  CE1FM3 BIT(32),
2 CE1FA4 BIT(32),                          /* FARW - 4  */
2  CE1FM4 BIT(32),
2 CE1FA5 BIT(32),                          /* FARW - 5  */
2  CE1FM5 BIT(32),
2 CE1FA6 BIT(32),                          /* FARW - 6  */
2  CE1FM6 BIT(32),
2 CE1FA7 BIT(32),                          /* FARW - 7  */
2  CE1FM7 BIT(32),
2 CE1FA8 BIT(32),                          /* FARW - 8  */
2  CE1FM8 BIT(32),
2 CE1FA9 BIT(32),                          /* FARW - 9  */
2  CE1FM9 BIT(32),
2 CE1FAA BIT(32),                          /* FARW - A  */
2  CE1FMA BIT(32),
2 CE1FAB BIT(32),                          /* FARW - B  */
2  CE1FMB BIT(32),
2 CE1FAC BIT(32),                          /* FARW - C  */
2  CE1FMC BIT(32),
2 CE1FAD BIT(32),                          /* FARW - D  */
2  CE1FMD BIT(32),
2 CE1FAE BIT(32),                          /* FARW - E  */
2  CE1FME BIT(32),
2 CE1FAF BIT(32),                          /* FARW - F  */
2  CE1FMF BIT(32),
2 CE1FAP BIN (31),            /* FARW FOR PROGRAM CALL  */
2 CE1FMP BIN (31),
2 CE1CR0 PTR,           /* CORE BLOCK REFENCE WORD - 0  */
2 CE1CT00 CHAR(1),
2 CE1CT01 BIT(8),
2 CE1CC0 BIN (15),
2 CE1CR1 PTR,           /* CORE BLOCK REFERENCE WORD - 1*/
2 CE1CT10 CHAR(1),
2 CE1CT11 BIT(8),
2 CE1CC1 BIN (15),
2 CE1CR2 PTR,           /* CORE BLOCK REFERENCE WORD - 2*/
2 CE1CT20 CHAR(1),
2 CE1CT21 BIT(8),
2 CE1CC2 BIN (15),
2 CE1CR3 PTR,           /* CORE BLOCK REFERENCE WORD - 3*/
2 CE1CT30 CHAR(1),
```

```
2 CE1CT31 BIT(8),
2 CE1CC3 BIN (15),
2 CE1CR4 PTR,          /* CORE BLOCK REFERENCE WORD - 4*/
2 CE1CT40 CHAR(1),
2 CE1CT41 BIT(8),
2 CE1CC4 BIN (15),
2 CE1CR5 PTR,          /* CORE BLOCK REFERENCE WORD - 5*/
2 CE1CT50 CHAR(1),
2 CE1CT51 BIT(8),
2 CE1CC5 BIN (15),
2 CE1CR6 PTR,          /* CORE BLOCK REFERENCE WORD - 6*/
2 CE1CT60 CHAR(1),
2 CE1CT61 BIT(8),
2 CE1CC6 BIN (15),
2 CE1CR7 PTR,          /* CORE BLOCK REFERENCE WORD - 7*/
2 CE1CT70 CHAR(1),
2 CE1CT71 BIT(8),
2 CE1CC7 BIN (15),
2 CE1CR8 PTR,          /* CORE BLOCK REFERENCE WORD - 8*/
2 CE1CT80 CHAR(1),
2 CE1CT81 BIT(8),
2 CE1CC8 BIN (15),
2 CE1CR9 PTR,          /* CORE BLOCK REFERENCE WORD - 9*/
2 CE1CT90 CHAR(1),
2 CE1CT91 BIT(8),
2 CE1CC9 BIN (15),
2 CE1CRA PTR,          /* CORE BLOCK REFERENCE WORD - A*/
2 CE1CTA0 CHAR(1),
2 CE1CTA1 BIT(8),
2 CE1CCA BIN (15),
2 CE1CRB PTR,          /* CORE BLOCK REFERENCE WORD - B*/
2 CE1CTB0 CHAR(1),
2 CE1CTB1 BIT(8),
2 CE1CCB BIN (15),
2 CE1CRC PTR,          /* CORE BLOCK REFERENCE WORD - C*/
2 CE1CTC0 CHAR(1),
2 CE1CTC1 BIT(8),
2 CE1CCC BIN (15),
2 CE1CRD PTR,          /* CORE BLOCK REFERENCE WORD - D*/
2 CE1CTD0 CHAR(1),
2 CE1CTD1 BIT(8),
2 CE1CCD BIN (15),
2 CE1CRE PTR,          /* CORE BLOCK REFERENCE WORD - E*/
2 CE1CTE0 CHAR(1),
2 CE1CTE1 BIT(8),
2 CE1CCE BIN (15),
2 CE1CRF PTR,          /* CORE BLOCK REFERENCE WORD - F*/
2 CE1CTF0 CHAR(1),
2 CE1CTF1 BIT(8),
2 CE1CCF BIN (15),
2 CE1CRP PTR,          /* CBRW FOR PROGRAM CALLS       */
2 CE1CTP CHAR (2),
2 CE1CCP BIN (15),
2 CE1FX0 CHAR (8),              /* FARW EXTENSION - 0  */
2 CE1FX1 CHAR (8),              /* FARW EXTENSION - 1  */
2 CE1FX2 CHAR (8),              /* FARW EXTENSION - 2  */
2 CE1FX3 CHAR (8),              /* FARW EXTENSION - 3  */
2 CE1FX4 CHAR (8),              /* FARW EXTENSION - 4  */
2 CE1FX5 CHAR (8),              /* FARW EXTENSION - 5  */
2 CE1FX6 CHAR (8),              /* FARW EXTENSION - 6  */
2 CE1FX7 CHAR (8),              /* FARW EXTENSION - 7  */
```

```
2 CE1FX8 CHAR (8),              /* FARW EXTENSION - 8  */
2 CE1FX9 CHAR (8),              /* FARW EXTENSION - 9  */
2 CE1FXA CHAR (8),              /* FARW EXTENSION - A  */
2 CE1FXB CHAR (8),              /* FARW EXTENSION - B  */
2 CE1FXC CHAR (8),              /* FARW EXTENSION - C  */
2 CE1FXD CHAR (8),              /* FARW EXTENSION - D  */
2 CE1FXE CHAR (8),              /* FARW EXTENSION - E  */
2 CE1FXF CHAR (8),              /* FARW EXTENSION - F  */
2 CE1SUD,                 /* DETAIL DATA LEVEL ERROR INDS */
  3 EBCSD0 BIT (8),       /* DATA LEVEL ERROR IND - 0     */
  3 EBCSD1 BIT (8),       /* DATA LEVEL ERROR IND - 1     */
  3 EBCSD2 BIT (8),       /* DATA LEVEL ERROR IND - 2     */
  3 EBCSD3 BIT (8),       /* DATA LEVEL ERROR IND - 3     */
  3 EBCSD4 BIT (8),       /* DATA LEVEL ERROR IND - 4     */
  3 EBCSD5 BIT (8),       /* DATA LEVEL ERROR IND - 5     */
  3 EBCSD6 BIT (8),       /* DATA LEVEL ERROR IND - 6     */
  3 EBCSD7 BIT (8),       /* DATA LEVEL ERROR IND - 7     */
  3 EBCSD8 BIT (8),       /* DATA LEVEL ERROR IND - 8     */
  3 EBCSD9 BIT (8),       /* DATA LEVEL ERROR IND - 9     */
  3 EBCSDA BIT (8),       /* DATA LEVEL ERROR IND - A     */
  3 EBCSDB BIT (8),       /* DATA LEVEL ERROR IND - B     */
  3 EBCSDC BIT (8),       /* DATA LEVEL ERROR IND - C     */
  3 EBCSDD BIT (8),       /* DATA LEVEL ERROR IND - D     */
  3 EBCSDE BIT (8),       /* DATA LEVEL ERROR IND - E     */
  3 EBCSDF BIT (8),       /* DATA LEVEL ERROR IND - F     */
2 CE1SUP,                 /* DETAIL PROGRAM LEVEL ERROR   */
  3 EBCSDP BIT (8),
2 CE1SUG,                 /* GROSS DATA LEVEL ERROR IND   */
  3 EBCSUG BIT (8),
2 CE1SP0 CHAR (6),        /* RESERVED                     */
2 CE1PL0 CHAR (8),        /* PROGRAM NESTING LEVEL - 0    */
2 CE1PL1 CHAR (8),        /* PROGRAM NESTING LEVEL - 1    */
2 CE1PL2 CHAR (8),        /* PROGRAM NESTING LEVEL - 2    */
2 CE1PL3 CHAR (8),        /* PROGRAM NESTING LEVEL - 3    */
2 CE1PL4 CHAR (8),        /* PROGRAM NESTING LEVEL - 4    */
2 CE1PL5 CHAR (8),        /* PROGRAM NESTING LEVEL - 5    */
2 CE1PL6 CHAR (8),        /* PROGRAM NESTING LEVEL - 6    */
2 CE1PL7 CHAR (8),        /* PROGRAM NESTING LEVEL - 7    */
2 CE1PL8 CHAR (8),        /* PROGRAM NESTING LEVEL - 8    */
2 CE1PL9 CHAR (8),        /* PROGRAM NESTING LEVEL - 9    */
2 CE1PLA CHAR (8),        /* PROGRAM NESTING LEVEL - A    */
2 CE1PLB CHAR (8),        /* PROGRAM NESTING LEVEL - B    */
2 CE1PLC CHAR (8),        /* PROGRAM NESTING LEVEL - C    */
2 CE1PLD CHAR (8),        /* PROGRAM NESTING LEVEL - D    */
2 CE1PLE CHAR (8),        /* PROGRAM NESTING LEVEL - E    */
2 CE1PLF CHAR (8),        /* PROGRAM NESTING LEVEL - F    */
2 CE1RDA BIN (31),        /* REGISTER SAVE AREA - RDA     */
2 CE1RDB BIN (31),        /* REGISTER SAVE AREA - RDB     */
2 CE1SVR BIN (31),        /* REGISTER SAVE AREA - RAC     */
2 CE1SV1 BIN (31),        /* REGISTER SAVE AREA - RG1     */
2 CE1SVA BIN (31),        /* REGISTER SAVE AREA - RGA     */
2 CE1SVB BIN (31),        /* REGISTER SAVE AREA - RGB     */
2 CE1SVC BIN (31),        /* REGISTER SAVE AREA - RGC     */
2 CE1SVD BIN (31),        /* REGISTER SAVE AREA - RGD     */
2 CE1SVE BIN (31),        /* REGISTER SAVE AREA - RGE     */
2 CE1SVF BIN (31),        /* REGISTER SAVE AREA - RGF     */
2 CE1SVP BIN (31),        /* REGISTER SAVE AREA - RAP     */
2 CE1PRL BIN (31),        /* PROGRAM LEVEL COUNT          */
2 CE1NST PTR,             /* ADDRESS OF PGM NESTING AREA  */
2 CE1CXR CHAR (8),        /* CONTROL TRANSFER FIELD       */
2 CE1CQE BIN (31),        /* 2305 COPY QUEUE CHAIN WORD   */
```

```
2 CE1SP1 CHAR (4),      /* RESERVED                      */
2 CE1SON CHAR (16),     /* SON DECODE FIELD              */
2 CE1AWA CHAR (8),      /* AUXILIARY WORK AREA           */
2 CE1AUT BIN (31),      /* AUTOMATIC STORAGE BLOCK       */
2 CE1PBI CHAR (2),      /* PROGRAM BASE ID               */
2 CE1DBI,               /* DATA BASE ID                  */
  3 CE1UID CHAR (2),
2 CE1SDBI CHAR (2),     /* SAVE DATA BASE ID             */
2 CE1SP7 CHAR (2),      /* RESERVED                      */
2 CE1TTA CHAR (8),      /* TEST TOOLS AREA               */
2 CE1COM,               /* CRPL                          */
  3 CE1DES PTR,         /* MESSAGE DESTINATION           */
  3 CE1ORG PTR,         /* MESSAGE ORIGIN                */
  3 CE1PRE CHAR (2),    /* CONSOLE PREFIX SMART ID       */
  3 CE1MID CHAR (2),    /* SMART USER ID                 */
  3 CE1CL1 BIT (8),     /* CONTROL INFORMATION           */
  3 CE1CL2 BIT (8),
  3 CE1CL3 BIT (8),
  3 CE1CL4 BIT (8),
  3 CE1CL5 BIT (8),
  3 CE1CL6 BIT (8),
  3 CE1CL7 BIT (8),
  3 CE1CL8 BIT (8),
2 CE1UTP BIT (8),       /* COMPOSITE USER - USER TYPE    */
2 CE1SUN BIT (8),       /* SUB USER NUMBER               */
2 CE1TNS BIT (8),       /* TTL NBR OF SUBUSERS           */
2 CE1SP9 BIT (8),       /* RESERVED                      */
2 CE1CPX,               /* CONTROL PROGRAM WORK AREA     */
  3 CE1CPA BIT (8),
  3 CE1CPB BIT (8),
  3 CE1CPC BIT (8),
  3 CE1CPD BIT (8),
2 CE1SP2 CHAR (4),      /* RESERVED                      */
2 CE1SP3 CHAR (6),      /* RESERVED                      */
2 CE1TRV BIN (15),      /* TRANSFER VECTOR FIELD         */
2 CE1PSW CHAR (8),      /* PROGRAM STATUS WORD           */
2 CE1IOC BIN (15),      /* INPUT/OUTPUT COUNTER          */
2 CE1HLD CHAR (1),      /* HOLD COUNTER                  */
2 CE1TAP CHAR (1),      /* SYMBOLIC TAPE MODULE NO.      */
2 CE1TIN BIT (16),      /* TAPE STATUS INDICATORS        */
2 CE1URM BIT (16),      /* UNIT RECORD                   */
2 CE1REC,               /* TEST TOOL OPTIONS             */
  3 CE1TOP BIT (8),     /* TRACE OPTIONS                 */
  3 CE1OUT,             /* TERMINAL ADDRESS              */
    4 EBROUT CHAR (3),
  3 CE1TST CHAR (4),    /* RECORDING/TEST                */
2 CE1SP4 CHAR (2),      /* RESERVED                      */
2 CE1SUC,               /* CONTROL PROGRAM FLAG          */
  3 EBCSUC BIT (8),
2 CE1SUI,               /* SYSTEM ERROR INDICATORS       */
  3 CE1SYE BIT (8),
2 CE1TRC PTR,           /* TRACE CNTRL XFER & CREATES    */
2 CE1TCU CHAR (1),      /* PTI CONTROL BYTE              */
2 CE1TIM CHAR (5),      /* ECB TIME STAMP                */
2 CE1IN1 BIT (8),       /* INDICATOR BYTE - 1            */
2 CE1RTT BIT (8),       /* TRACE OUTPUT SELECTION        */
2 CE1SP5 CHAR (8),      /* RESERVED                      */
2 CE1ARS,               /* USER REGISTER SAVE AREA       */
  3 CE1URA BIN (31),    /* REGISTER SAVE AREA - RDA      */
  3 CE1URB BIN (31),    /* REGISTER SAVE AREA - RDB      */
  3 CE1UR0 BIN (31),    /* REGISTER SAVE AREA - RAC      */
```

```
    3 CE1UR1 BIN (31),    /* REGISTER SAVE AREA - RG1     */
    3 CE1UR2 BIN (31),    /* REGISTER SAVE AREA - RGA     */
    3 CE1UR3 BIN (31),    /* REGISTER SAVE AREA - RGB     */
    3 CE1UR4 BIN (31),    /* REGISTER SAVE AREA - RGC     */
    3 CE1UR5 BIN (31),    /* REGISTER SAVE AREA - RGD     */
    3 CE1UR6 BIN (31),    /* REGISTER SAVE AREA - RGE     */
    3 CE1UR7 BIN (31),    /* REGISTER SAVE AREA - RGF     */
  2 CE1WKB,               /* WORK AREA 2                  */
    3 EBX000F,
      4 (EBX000,EBX001,EBX002,EBX003) CHAR (1),
    3 EBX004F,
      4 (EBX004,EBX005,EBX006,EBX007) CHAR (1),
    3 EBX008F,
      4 (EBX008,EBX009,EBX010,EBX011) CHAR (1),
    3 EBX012F,
      4 (EBX012,EBX013,EBX014,EBX015) CHAR (1),
    3 EBX016F,
      4 (EBX016,EBX017,EBX018,EBX019) CHAR (1),
    3 EBX020F,
      4 (EBX020,EBX021,EBX022,EBX023) CHAR (1),
    3 EBX024F,
      4 (EBX024,EBX025,EBX026,EBX027) CHAR (1),
    3 EBX028F,
      4 (EBX028,EBX029,EBX030,EBX031) CHAR (1),
    3 EBX032F,
      4 (EBX032,EBX033,EBX034,EBX035) CHAR (1),
    3 EBX036F,
      4 (EBX036,EBX037,EBX038,EBX039) CHAR (1),
    3 EBX040F,
      4 (EBX040,EBX041,EBX042,EBX043) CHAR (1),
    3 EBX044F,
      4 (EBX044,EBX045,EBX046,EBX047) CHAR (1),
    3 EBX048F,
      4 (EBX048,EBX049,EBX050,EBX051) CHAR (1),
    3 EBX052F,
      4 (EBX052,EBX053,EBX054,EBX055) CHAR (1),
    3 EBX056F,
      4 (EBX056,EBX057,EBX058,EBX059) CHAR (1),
    3 EBX060F,
      4 (EBX060,EBX061,EBX062,EBX063) CHAR (1),
    3 EBX064F,
      4 (EBX064,EBX065,EBX066,EBX067) CHAR (1),
    3 EBX068F,
      4 (EBX068,EBX069,EBX070,EBX071) CHAR (1),
    3 EBX072F,
      4 (EBX072,EBX073,EBX074,EBX075) CHAR (1),
    3 EBX076F,
      4 (EBX076,EBX077,EBX078,EBX079) CHAR (1),
    3 EBX080F,
      4 (EBX080,EBX081,EBX082,EBX083) CHAR (1),
    3 EBX084F,
      4 (EBX084,EBX085,EBX086,EBX087) CHAR (1),
    3 EBX088F,
      4 (EBX088,EBX089,EBX090,EBX091) CHAR (1),
    3 EBX092F,
      4 (EBX092,EBX093,EBX094,EBX095) CHAR (1),
    3 EBX096F,
      4 (EBX096,EBX097,EBX098,EBX099) CHAR (1),
    3 EBX100F,
      4 (EBX100,EBX101,EBX102,EBX103) CHAR (1),
    3 EBX104F,
```

```
            4 (EBXSW0,EBXSW1,EBXSW2,EBXSW3) BIT (8),
          3 EBX108F,
            4 (EBXSW4,EBXSW5,EBXSW6,EBXSW7) BIT (8),
        2 CE1SP6 CHAR (15),     /* RESERVED                    */
        2 CE1FLG BIT (8);       /* FORMAT FLAG                 */
/*      2 CE1RES CHAR (300),       RESERVED SYSTEM AREA       */
/*      2 CE1USA CHAR (336),       USER AREA                  */
DECLARE 1 EBCFW0 DEFINED CE1FA0,
        2 EBCID0 BIN (15),
        2 EBCRC0 CHAR (1),
        2 EBCCN0 CHAR (1),
        2 EBCFA0,
          3 EBCFM0 CHAR (1),
          3 EBCFC0 CHAR (1),
          3 EBCFH0 CHAR (1),
          3 EBCFR0 CHAR (1);
DECLARE 1 EBCFW1 DEFINED CE1FA1,
        2 EBCID1 BIN (15),
        2 EBCRC1 CHAR (1),
        2 EBCCN1 CHAR (1),
        2 EBCFA1,
          3 EBCFM1 CHAR (1),
          3 EBCFC1 CHAR (1),
          3 EBCFH1 CHAR (1),
          3 EBCFR1 CHAR (1);
DECLARE 1 EBCFW2 DEFINED CE1FA2,
        2 EBCID2 BIN (15),
        2 EBCRC2 CHAR (1),
        2 EBCCN2 CHAR (1),
        2 EBCFA2,
          3 EBCFM2 CHAR (1),
          3 EBCFC2 CHAR (1),
          3 EBCFH2 CHAR (1),
          3 EBCFR2 CHAR (1);
DECLARE 1 EBCFW3 DEFINED CE1FA3,
        2 EBCID3 BIN (15),
        2 EBCRC3 CHAR (1),
        2 EBCCN3 CHAR (1),
        2 EBCFA3,
          3 EBCFM3 CHAR (1),
          3 EBCFC3 CHAR (1),
          3 EBCFH3 CHAR (1),
          3 EBCFR3 CHAR (1);
DECLARE 1 EBCFW4 DEFINED CE1FA4,
        2 EBCID4 BIN (15),
        2 EBCRC4 CHAR (1),
        2 EBCCN4 CHAR (1),
        2 EBCFA4,
          3 EBCFM4 CHAR (1),
          3 EBCFC4 CHAR (1),
          3 EBCFH4 CHAR (1),
          3 EBCFR4 CHAR (1);
DECLARE 1 EBCFW5 DEFINED CE1FA5,
        2 EBCID5 BIN (15),
        2 EBCRC5 CHAR (1),
        2 EBCCN5 CHAR (1),
        2 EBCFA5,
          3 EBCFM5 CHAR (1),
          3 EBCFC5 CHAR (1),
          3 EBCFH5 CHAR (1),
          3 EBCFR5 CHAR (1);
```

```
DECLARE 1 EBCFW6 DEFINED CE1FA6,
          2 EBCID6 BIN (15),
          2 EBCRC6 CHAR (1),
          2 EBCCN6 CHAR (1),
          2 EBCFA6,
            3 EBCFM6 CHAR (1),
            3 EBCFC6 CHAR (1),
            3 EBCFH6 CHAR (1),
            3 EBCFR6 CHAR (1);
DECLARE 1 EBCFW7 DEFINED CE1FA7,
          2 EBCID7 BIN (15),
          2 EBCRC7 CHAR (1),
          2 EBCCN7 CHAR (1),
          2 EBCFA7,
            3 EBCFM7 CHAR (1),
            3 EBCFC7 CHAR (1),
            3 EBCFH7 CHAR (1),
            3 EBCFR7 CHAR (1);
DECLARE 1 EBCFW8 DEFINED CE1FA8,
          2 EBCID8 BIN (15),
          2 EBCRC8 CHAR (1),
          2 EBCCN8 CHAR (1),
          2 EBCFA8,
            3 EBCFM8 CHAR (1),
            3 EBCFC8 CHAR (1),
            3 EBCFH8 CHAR (1),
            3 EBCFR8 CHAR (1);
DECLARE 1 EBCFW9 DEFINED CE1FA9,
          2 EBCID9 BIN (15),
          2 EBCRC9 CHAR (1),
          2 EBCCN9 CHAR (1),
          2 EBCFA9,
            3 EBCFM9 CHAR (1),
            3 EBCFC9 CHAR (1),
            3 EBCFH9 CHAR (1),
            3 EBCFR9 CHAR (1);
DECLARE 1 EBCFWA DEFINED CE1FAA,
          2 EBCIDA BIN (15),
          2 EBCRCA CHAR (1),
          2 EBCCNA CHAR (1),
          2 EBCFAA,
            3 EBCFMA CHAR (1),
            3 EBCFCA CHAR (1),
            3 EBCFHA CHAR (1),
            3 EBCFRA CHAR (1);
DECLARE 1 EBCFWB DEFINED CE1FAB,
          2 EBCIDB BIN (15),
          2 EBCRCB CHAR (1),
          2 EBCCNB CHAR (1),
          2 EBCFAB,
            3 EBCFMB CHAR (1),
            3 EBCFCB CHAR (1),
            3 EBCFHB CHAR (1),
            3 EBCFRB CHAR (1);
DECLARE 1 EBCFWC DEFINED CE1FAC,
          2 EBCIDC BIN (15),
          2 EBCRCC CHAR (1),
          2 EBCCNC CHAR (1),
          2 EBCFAC,
            3 EBCFMC CHAR (1),
            3 EBCFCC CHAR (1),
```

```
            3 EBCFHC CHAR (1),
            3 EBCFRC CHAR (1);
DECLARE 1 EBCFWD DEFINED CE1FAD,
          2 EBCIDD BIN (15),
          2 EBCRCD CHAR (1),
          2 EBCCND CHAR (1),
          2 EBCFAD,
            3 EBCFMD CHAR (1),
            3 EBCFCD CHAR (1),
            3 EBCFHD CHAR (1),
            3 EBCFRD CHAR (1);
DECLARE 1 EBCFWE DEFINED CE1FAE,
          2 EBCIDE BIN (15),
          2 EBCRCE CHAR (1),
          2 EBCCNE CHAR (1),
          2 EBCFAE,
            3 EBCFME CHAR (1),
            3 EBCFCE CHAR (1),
            3 EBCFHE CHAR (1),
            3 EBCFRE CHAR (1);
DECLARE 1 EBCFWF DEFINED CE1FAF,
          2 EBCIDF BIN (15),
          2 EBCRCF CHAR (1),
          2 EBCCNF CHAR (1),
          2 EBCFAF,
            3 EBCFMF CHAR (1),
            3 EBCFCF CHAR (1),
            3 EBCFHF CHAR (1),
            3 EBCFRF CHAR (1);
DECLARE 1 EBCCX0 DEFINED CE1CR0,
          2 EBCCR0 PTR,
          2 EBCCT0 CHAR (2),
          2 EBCCC0 BIN (15);
DECLARE 1 EBCCX1 DEFINED CE1CR1,
          2 EBCCR1 PTR,
          2 EBCCT1 CHAR (2),
          2 EBCCC1 BIN (15);
DECLARE 1 EBCCX2 DEFINED CE1CR2,
          2 EBCCR2 PTR,
          2 EBCCT2 CHAR (2),
          2 EBCCC2 BIN (15);
DECLARE 1 EBCCX3 DEFINED CE1CR3,
          2 EBCCR3 PTR,
          2 EBCCT3 CHAR (2),
          2 EBCCC3 BIN (15);
DECLARE 1 EBCCX4 DEFINED CE1CR4,
          2 EBCCR4 PTR,
          2 EBCCT4 CHAR (2),
          2 EBCCC4 BIN (15);
DECLARE 1 EBCCX5 DEFINED CE1CR5,
          2 EBCCR5 PTR,
          2 EBCCT5 CHAR (2),
          2 EBCCC5 BIN (15);
DECLARE 1 EBCCX6 DEFINED CE1CR6,
          2 EBCCR6 PTR,
          2 EBCCT6 CHAR (2),
          2 EBCCC6 BIN (15);
DECLARE 1 EBCCX7 DEFINED CE1CR7,
          2 EBCCR7 PTR,
          2 EBCCT7 CHAR (2),
          2 EBCCC7 BIN (15);
```

```
DECLARE 1 EBCCX8 DEFINED CE1CR8,
          2 EBCCR8 PTR,
          2 EBCCT8 CHAR (2),
          2 EBCCC8 BIN (15);
DECLARE 1 EBCCX9 DEFINED CE1CR9,
          2 EBCCR9 PTR,
          2 EBCCT9 CHAR (2),
          2 EBCCC9 BIN (15);
DECLARE 1 EBCCXA DEFINED CE1CRA,
          2 EBCCRA PTR,
          2 EBCCTA CHAR (2),
          2 EBCCCA BIN (15);
DECLARE 1 EBCCXB DEFINED CE1CRB,
          2 EBCCRB PTR,
          2 EBCCTB CHAR (2),
          2 EBCCCB BIN (15);
DECLARE 1 EBCCXC DEFINED CE1CRC,
          2 EBCCRC PTR,
          2 EBCCTC CHAR (2),
          2 EBCCCC BIN (15);
DECLARE 1 EBCCXD DEFINED CE1CRD,
          2 EBCCRD PTR,
          2 EBCCTD CHAR (2),
          2 EBCCCD BIN (15);
DECLARE 1 EBCCXE DEFINED CE1CRE,
          2 EBCCRE PTR,
          2 EBCCTE CHAR (2),
          2 EBCCCE BIN (15);
DECLARE 1 EBCCXF DEFINED CE1CRF,
          2 EBCCRF PTR,
          2 EBCCTF CHAR (2),
          2 EBCCCF BIN (15);
DECLARE CITPCA PTR DEFINED EBW100F;
```

APPENDIX B.

SPECIAL SABRETALK CONSIDERATIONS FOR TPFDF USERS

As a general rule, assembler macros within a SABRETALK segment are simply passed along to the generated assembler language program. Any processing by the compiler is normally limited to syntax checking of the macro. Consequently, if macro parameters contain entries in the form “keyword=value", these parameters have to be enclosed in quotes so as not to cause syntax errors when examined by the syntax checker. Example:

```
ROUTC 'LIST=R2','LEV=D2'(#R2=LISTPTR);
```

In the case of TPFDF macros, however, special processing has been introduced to allow more flexible interchange of information between items known to SABRETALK and those needed by TPFDF. As such, any keyword parameters (of the form "keyword=value") that are not enclosed in quotes, will be analyzed by the compiler for possible resolution of SABRETALK symbols. For example:

```
TPFDF ADD,'FILE=GR20SR','NEWLREC=R5',ERROR=ERR1(#R5=P);
```

would cause the compiler to pass the first three parameters (**ADD, FILE=GR20SR, NEWLREC=R5**) exactly as shown (except for the delimiting quote marks), while the fourth parameter (**ERROR=ERR1**) would be changed to read “**ERROR=ERR1$** “. (Assuming that the compiler option **DOLLAR** is in effect.)

In addition, because TPFDF may have to know about some fields which have been defined to the compiler (but are not generally given to the assembler in base displacement formats), the compiler will generate the correct references to such fields if a special flag ("**?**") is coded immediately before each field. For example:

```
TPFDF READ,'FILE=GR20SR',UP,
      KEY1=('R=GR20KEY',S=?P->GR20KEY),
      KEY2=('R=GR20AL2',S=?P->GR20AL2),ERROR=ERR1;
```

In the above case, the sub-parameters **S=?**... cause the symbols given to be substituted with expressions that yield the correct base and displacements so that the correct locations can be identified at assembly time. In addition, the compiler will also cause the appropriate register to be loaded with the correct base address so that the code will execute as expected. As such, the above macro reference would cause the following code to be generated:

```
L     R4,P$(R7)
TPFDF READ,FILE=GR20SR,UP,
      KEY1=(R=GR20KEY,S=GR20KEY$(R4)),
      KEY2=(R=GR20AL2,S=GR20AL2$(R4)),ERROR=ERR1$
```

The compiler automatically selects the registers in a predetermined order starting with R8 and continuing "downward" to R1 if necessary. Thus:

- R9 will be selected for any fields in ENTRYBLOCK storage (ECB).
- R7 will be selected for any fields in AUTOMATIC storage.
- R6 will be selected for any fields of BASED storage assigned a permanent base of R6.
- R5 will be selected for any fields of BASED storage assigned a permanent base of R5.
- R4 thru R1 will be selected and loaded for any other fields as needed.

Programmers should keep these assignments in mind whenever they need to pass values to TPFDF in registers, i.e., they should not attempt to pass an automatic storage field while at the same time trying to load R7 with some other value. It is best for programmers to select registers starting at R0 and working "upward" as far as necessary to avoid conflicts with the compiler selection.

Because **CONSTANT** storage fields reside in the same core block as the program, and consequently are addressed through R8, they should not be passed to the TPFDF macro.

USER MACROS

This category is for Macros not specifically written for SABRETALK use which may not preserve registers R1 - R7. Also, all coding used for Special macros (TPFDF type) will be allowed. For example, use of a Question Mark to pass Base and Displacement.

After this type of Macro is issued, R7 will be restored from the ECB (CE1AUT) and R1 - R6 will be marked as altered. Example:

```
TESTC   ADD,'REF=SW0SEA',UP,WAIT=TAG1,
        REG=?CORD6,ERROR=TAG2
        (CORD5=#R1,#R2=CP_ORD5);
```

generates:

```
        L     R2,CP$ORD5$(R5)
        TESTC    ADD,REF=SW0SEA,UP,WAIT=TAG1$,REG=CORD6$(R6),          X
                 ERROR=TAG2$
        L     R7,CE1AUT
        LA    R7,8(,R7)
        L     R5,CPINCPTR(R7)
        L     R6,$PPTR$(R7)
        ST    R1,CORD5$(R6)
```

GLOSSARY

Definitions are biased to reflect a particular meaning associated with SABRETALK.

An asterisk (*) is used to mark duplicate entries: information about ''Address, Absolute *'' can be found under ''Absolute Address''

Absolute Address
The number of each storage location which is permanently wired into the hardware by the manufacturer. In the case of core memory this is a byte address.

Absolute Value
A number whose magnitude is given but whose sign is not.

Addition, File *

Address, Absolute *

Address, Base *

Address, Machine *

Address Modification
The incrementation of an address for the purpose of preserving the relativity of relocated programs.

Address, Relative *

ALGOL
An acronym for ''Algorithmic Language'', one of the more common compilers. ALGOL is a powerful statement-oriented algebraic language used principally for scientific problems. It was the model for many of the newer artificial computer languages.

Aligned
A statement keyword that allows areas to be positionally adjusted, in byte multiples, so that they can be more readily accessed and so as to more closely conform to the format (doubleword, full-word, half-word, byte) of a major structure. (See **Packed**)

Allocation
The assignment of specific memory locations to program instructions or data.

Allocation, Dynamic *

Allocation, Storage *

Alphabet
An ordered set of language characters. The Roman alphabet consists of the 26 letters, a through z, while the Universal alphabet of 29 characters contains the 26 letters and the symbols **$, @, #**.

Alphanumeric
An ordered set of characters that consists of the characters of an alphabet and numerals.

And, Logical *

And Symbol
The ampersand symbol (&) used to represent the logical ''and'' operator.

Area
A portion of memory whose smallest unit is a bit value and within which the binary representation of a value or values may reside.

Arithmetic, Boolean *

Arithmetic Operator
One of the symbols for addition, subtraction, multiplication, and division (**+,-,*,/**). Used to indicate the mathematical treatment of one operand by another.

Array
A collection of logically related data items, all of which have identical data attributes.

Assemble
The act of producing machine coding from a representative, symbolic version.

Assembler
A program designed to produce machine language coding from a representative, symbolic version.

Assembler Code
The specification of program instructions and data in symbolic form. (also **Assembler coding)**

Assembler Instruction
An element of Assembler code normally representative of a single machine instruction.

Assembler Language
(see **Assembler Code**)

Assignment Operator
The character (**=**) that signifies that an assignment of data is to be performed.

Assignment Statement
An executable SABRETALK instruction used to assign a value into an area or field. The assignment operator is an equal sign (**=**).

Asterisk
Used as an operator, this character (*****) denotes multiplication.

At Sign
Used as a character in the Universal alphabet, this character (**@**) is used as a prefix for a global tag.

Attribute
A characteristic of a value or of an area. Attributes in SABRETALK are frequently keywords.

Attribute, Data *

BAL
An acronym for Basic Assembler Language, an Assembler language used for many computers and which is the object code or output code of SABRETALK.

Base
The starting location of data. Also, the first digit of a counting system; for example, when counting items, if the first, second and third items are numbered 0, 1, 2 then the base is zero; if they are numbered 1,2,3 then the base is one. In SABRETALK arrays are subscripted base one. (See **Radix**.)

Base Address
Generally, the absolute address of the start of some area, to which may be added a relative address. A choice of relative and/or absolute addresses provides a more flexible programming environment.

Binary

Pertaining to a number system that uses only two digits: zero and one. Values can be represented by digit groups: the decimal number "three" by 000011 or by 11; the letter "U" by 11100101. Arithmetic is performed by decimal hardware or by binary hardware according to convention.

Binary Code

The representation of data in terms of binary digits.

Binary Coded Decimal

The representation of decimal digits by one of a number of binary code sets.

Binary Digit

A character representing one of the two possible binary values; zero or one.

Binary Number

A representation of a number by positional notation of binary digits. Where a point (`.`) is used to separate the integer portion and fractional portion of a number, the binary number 111.01 would represent:

```
+  1  times  2  raised-to-the-power  two
+  1    ''   2         ''            one
+  1    ''   2         ''            zero
+  0    ''   2         ''            minus one
+  1    ''   2         ''            minus two
```

Bit

An acronym meaning "**BI**nary digi**T**", the smallest unit of computer data. Frequently used to also refer to the area in which a binary digit may reside.

Blank

The representation of:

- the absence of print where a print character might appear.
- the absence of a punch where a punch character might appear.
- a specific symbol representing the absence of a print or punch character.

In SABRETALK a blank is considered a character.

Body, Statement *

Boolean

Pertaining to the algebra of logic formulated by George Boole; by extension, to the method whereby the truth or falsity of statements may be represented by the binary digits zero and one: conclusions from ordered sets of statements may then be derived by the rules of Boolean algebra.

Boolean Arithmetic

The operations involved in the use of the logical operators "and", "or" and "not".

Boundary

A restriction that limits a core location to a modular one.

Boundary, Byte *

Boundary, Character *

Boundary, Double-word *

Boundary, Full-word *

Boundary, Half-word *

Branch
One of the paths chosen as a result of the execution of a decision instruction; to give control to an instruction other than the next sequential one.

Branch, Conditional *

Break Character
An underline character (_) frequently used as an aid in visually partitioning identifiers or labels.

Built-in Function
A SABRETALK keyword that results in the performance of a specific task through the collective action of an instruction set.

Byte
An eight-bit modular unit of serial core represented by a core address.

Byte Boundary
A restriction that limits a core address to a byte module; also a character module.

Call
To transfer control to a specified closed internal procedure.

Card Column
A vertical strip through 1/80th of a card. Twelve rows cross the strip giving 4096 possible values per column.

Card Image
A representation of an eighty-character record that constitutes a method of standardizing the size of record modules.

Card Format
A string of eighty consecutive columns where each column could contain one character.

Character
A single planar graphic used as a visual representation of data. In a computer a character is represented by an eight-bit module. Since it can also represent the addressable element of core, it is often referenced as a byte. A blank is to be considered a character in SABRETALK.

Character Boundary
A byte boundary.

Character, Break *

Character, Drifting *

Character, Special *

Code
To express information in character patterns. The character patterns that express information.

Code, Assembler *

Code, Binary *

Code, Compiler *

Code, Decimal *

Code, Machine *

Code, Object *

Code, Re-entrant
(see **Re-entrant**)

Code, Source *

Column, Card *

Command
An instruction.

Comment
An aside. A note or remark that accompanies a portion of source code and has no effect on object coding. SABRETALK comments have the form **`/*comment*/`**.

Comparison Operator
Any of the symbols (**`=  <  >  ^=  ^<  ^>  <=  >=`**) that signify a comparison is to be performed.

Compile
To translate from a more sophisticated, higher-level language to a more simple, machine-oriented language.

Compiler
A program that compiles. Input to a Compiler (source code) is manipulated so as to produce a more machine-compatible version of the data, data-structures and procedures desired. The consequential output (object code) is usually accompanied by such programmer aids as cross- references, error messages, etc.

Compiler Code
(see **Compiler Language**)

Compiler Instruction
(see **Compiler Statement**)

Compiler Language
The specification of program instructions and data in a more general problem-oriented statement form, rather than the strict machine-oriented instruction format of an Assembler language.

Compiler Statement
In SABRETALK, a basic language element that ends in a semicolon (**`;`**) and that is used to **`DECLARE`** data and/or specify executable directives.

Complement
The difference between a number and a string of identical digits; for example, the complement of 123 (using the digit string 99999) is 99876.

Composite
A set of two or more consecutive characters. In SABRETALK, composites may not contain a blank.

Composite Operator
A set of two or more consecutive characters that signifies an arithmetic, logical, relational, parenthetical or concatenation operation is to be performed.

Computer Program
The orderly set of instructions, associative data and work area(s) needed to perform a computer task.

Concatenation
The orderly appending of sequential data to sequential data. The result, of course, is a new and larger set of data.

Concatenation Operator
The composite (**| |**) that signifies a concatenation operation is to be performed.

Concatenation Symbol
A composite of two vertical bars (**| |**) used to represent the concatenation operator.

Conditional Branch
The giving of control to an instruction as a result of executing a test instruction.

Constant
A data item, referenced by an identifier, the value of which may NOT change during execution of the program.

Constant String
An ordered sequence of constants. In SABRETALK, the individual selection of constants from a string is facilitated by declaring the area(s) in which they reside to be an array.

Control Program
A program or routine that supervises all other programs and is responsible for loading them, performing their I/O (input-output) tasks, for initiating and monitoring their activity and for terminating them. The control routine for programs produced by SABRETALK is called TPF (Transaction Processing Facility).

Control Routine
(see **Control Program**)

Core Memory
The large area of computer hardware used for the prime storage of data, instructions, programs etc. in a computer.

Data
Representations of information. In SABRETALK, the basic data unit is a bit from which are built elements called bytes, half-words, etc.. Though instructions are data elements, the SABRETALK guide favors the term "instruction" for commands and "data" for non-instructional coding.

Data Attribute
A characteristic of a value or of an area in which a value may be found, not the value itself.

Data Field
An area in which information is to reside.

Data Item
Generally, a singular value.

Data LABEL
A misnomer; in the SABRETALK guide an attempt is made to refer to instruction names as "LABELS" and to data names as "IDENTIFIERS".

Data Organization
The arrangement of data so that elements may be more easily or quickly accessed. Sequencing, indexing and partitioning are some of the methods of arranging data.

Data Processing
Any of the operations performed on data by a computer.

Data Record
A collection of data fields or data items that have some correlation.

Data Set
Specifically, any data collection; by context, a collection of records.

Data Statement
In SABRETALK, the term used to identify a statement that particularly defines (declares) a data item.

Decimal, Binary-coded *

Decimal Code
The representation of numeric information using decimal digits.

Decimal Digit
One of the characters, zero (0) through nine (9), used to represent the numbers of a system of radix 10 (the decimal system.)

Decimal Point
A period (**.**) used when necessary to define the boundary of the integer and/or fractional of a number.

Decision Table
A two-dimensional chart used to format the results of logical operations. A truth table for a logical 'AND' would be:

```
------------------------------
if A is true (1)             |
          -------------      |
if A is false(/)       |     |
----------------+-----+-----|
if B is true    |  /  |  1  |
          -------+-----+-----|
if B is false   |  /  |  /  |
----------------+-----+-----
```

Delimiter
An indicator that restricts sequential extension into another area.

Diagram, Logic
(see **Decision Table**)

Digit
A character used to denote a unit multiple. Hexadecimal digits, for example, consist of the digits 0 through F

Digit, Binary *

Digit, Decimal *

Digit, Hexadecimal *

Digit, Significant *

Dimension
An attribute declared for arrays that specifies the number of identical sub-units in the array, terminated by an **END** statement:

DO Group

In SABRETALK a relational set of statements or a part of a relational statement. The **DO** group begins with a **DO** statement and is terminated by an **END** statement:

```
a: DO b = 1 TO 9 BY 2 WHILE c = 5; ...END;
IF d = f THEN DO g = h; END;
```

DO Statement

In SABRETALK, the first statement in a DO group.

Double-word

An eight-byte unit of contiguous core on a double-word boundary.

Double-word Boundary

A byte boundary whose address is a multiple of 8.

Drifting Character

In SABRETALK, an editing character the positioning of which depends upon the location of the most significant digit of an edited number.

Dynamic Allocation

The act of providing core areas at the request of, and during the run time of, a program.

Edit

To peruse and/or modify the format or the content of data. In SABRETALK, editing is normally restricted to the modification of output data, particularly data that is in zoned-decimal format. The prime purpose of editing is to produce printable or displayable numeric data.

Element

A basic or fundamental portion of data.

Element, Language *

Enclosure

A pair of delimiters that surround a sequential field.

Entry Point

In SABRETALK, the entry point of a program can be considered to be the first **PROC** statement in the program. From here, control will pass to the **START** statement, if one is present, or will flow.

Equal Sign

The character (**=**) used in SABRETALK for assignment and comparison of variables.

Exclusive Or

A logical add operation in which a statement is assigned a value of one if true, zero if false. The sum (no carry) of the values assigned to each of a set of statements determines the truthfulness of the set.

Executive Routine

(see **Control Program**)

Expression

Two or more values and the enclosed operator(s) that constitute the arithmetic, logical, relational or concatenation operations.

Expression, Logical *

External Reference

That which refers to an area outside of the program.

Factor

A multiplier or an integral divisor.

Factoring of Terms

In SABRETALK, the combining of statements with similar attributes into one statement. Only one level of a structure may be factored:

```
DCL (a,b,c,d) BIN;
```

Field

A part of an area or a part of a record. Data is frequently structured in the following largest-to-smallest graduations: library, data set, record, word, field.

Field, Data *

File Addition

The concatenation of files.

Fixed Point

An arithmetic in which the (decimal) point is placed at a specific position in a field and is not moved thereafter. In contrast, floating point is an arithmetic in which the point is positioned or repositioned so that a maximum amount of significant digits may be preserved and so that one (usually) digit precedes the point.

Format, Card *

Full-word

A four-byte unit of contiguous core on a full-word boundary.

Full-word Boundary

A byte boundary whose address is a multiple of 4.

Function

- An activity.
- The way in which an activity is performed.
- In SABRETALK, a specific term for an internal procedure to which data is passed and which returns to the caller a computed result.

Function, Built-in *

Group, `DO` *

Half-word

A two-byte unit of contiguous core on a half-word boundary.

Half-word Boundary

A byte boundary whose address is an even number.

Hexadecimal Code

The representation of numeric information using hexadecimal digits.

Hexadecimal Digit

One of the characters used to represent numbers in the number system of radix 16 (the hexadecimal system.) The digits are represented by the characters **`0`** through **`9`** and by the letters **`A`** through **`F`**. The binary equivalents of the sixteen digits are **`0000`** through **`1111`**.

Identifier

In SABRETALK, the name given to a value or to a data field. The term “LABEL” will be generally limited, in this guide, to the name of an executable statement.

Identifying Keyword
A composite reserved in SABRETALK for defining data characteristics or for specifying the function of a statement. Keywords are classified as statement identifiers, data attributes, separators and built-in function names.

`IF` Statement
In SABRETALK, a relational statement that provides "then" or "else" branch control when the results of a test are true or false.

Image, Card *

Inclusive Or
A logical overlay operation in which statements are assigned a value of one if true and a value of zero if false. The logical resultant of OR'ing the statements is true unless all of the statements are false. The SABRETALK "OR" operation is an inclusive "OR".

Index
The integer rank of an item in a sequential set. In SABRETALK, sets of similar data elements are called arrays and an element is referenced by subscript, base 1, thus:

E(1) = first element (named "E") of array "A"
E(2) = second element of array "A"
E(3) = third element of array "A"

Infix
In SABRETALK, the type of operator that is inserted between two operands, as contrasted to the type of operator (prefix) that precedes one operand. All operators in SABRETALK are infix. The two operators plus (**+**) and minus (**-**) may also appear as 'prefix' operators.

Information
That portion of a signal or of a communication that has meaning.

Initialize
To bring to, or bring back to a starting state. SABRETALK will provide, for example, the object code necessary to initialize "**`DO`**" loops when they are entered or re-entered.

Input
In relation to programs produced by SABRETALK, input would be any data supplied to the program, before or after initialization, by a source external to the program.

Instruction
The command, or the equivalent of a command, that can cause programmable computer activity. Instructions are usually:

- executed by the central processing unit.
- composed of binary numbers.
- represented as an operation to be performed and the operand(s) to be operated upon.

Instructions of higher-level languages are called statements.

Instruction, Assembler *

Instruction, Compiler *
(see **Compiler Statement**)

Instruction, Machine *

Item
A singular portion of data that conveys unit information. In a general hierarchy of data structure, an item might be a bit, a field, a word, etc..

Item, Data *

Iteration
One pass through a series of instructions, a series usually designed to manage modifications at each of a number of passes. In SABRETALK, **`IF`** statements, **`DO`** statements and **`GOTO`** statements provide for iterative processing.

I/O
Input and/or output. (see **Input**, **Output**)

Keyword
In SABRETALK, a composite reserved for defining data characteristics or for specifying the function of a statement. Keywords are classed as data attributes, statement identifiers, separating keywords and built-in- function names.

Keyword, Identifying *

Keyword, Separating
(see **Separator**)

Label
The name given to an executable statement commonly equated with a core address. The SABRETALK guide attempts to distinguish between names of executable statements (labels) and names of values or data areas (identifiers.)

Label, Program *

Label, Statement *

Language
The syntax, the rules, the set of elements that comprise a communication system.

Language, Assembler *

Language, Compiler *

Language Element
A unit portion of a language; a character, a composite, an operator, an expression, a procedure, etc.

Language, Machine *

Length
An attribute of linear dimension, in terms of bits, bytes, etc. A distinction should be made between length and precision. Length applies strictly to measurement; precision refers to the accuracy with which a quantity is represented, particularly:

- when **`DECIMAL`** fractions are converted to **`BINARY`** fractions and vice-versa.
- when arithmetic operations result in truncation or in rounding.
- when a value is moved from a larger to a smaller area and some significance is lost.

Letters, Lower-case *

Letters, Upper-case *

List
A set of data items that are to be printed, read or displayed in order. In most cases, items in a list are to be treated serially from beginning to end.

Literal
A data item which does not have an identifier, and which is referenced by the actual value it represents.

Location
The specification of a place into which data may be stored. A more general term than the word 'address'.

Logical And
A logical operation in which statements are assigned a value of zero if false, a non-zero value if true and then the product (a bit multiplication product with no carry) is used to determine whether the set of statements are true (1) or false (0).

Logical Expression
A portion of a statement that indicates one or more logical operations is to be performed.

Logical Not
A negating operation in which a statement is assigned a 'true' value if false and assigned a 'false' value if true. In SABRETALK, the 'not' symbol (**^**) is used as:

- a logical negate where the essence of the result is zero or nonzero (true or false).

e.g.:

```
IF ^ A THEN GOTO B ;
```

- an arithmetic negate where each bit in a location is replaced with a bit of opposite value. **NOTE**: This is not the same as changing the sign.

Logical Operator
A symbol that indicates a logical operation is to be performed. The three logical operators in SABRETALK are

- the 'not' symbol (**^**) - a logical not
- the 'and' symbol (**&**) -a logical and
- the "or" symbol (**|**) - an inclusive or

Logical Or
(see **Inclusive Or**)

Lower-case Letters
The non-capital constructs of the Roman alphabet.

Machine Address
A unique number assigned to a small module of memory.

Machine Code
The character pattern(s) used to represent commands or data in a computer; machine language code.

Machine Instruction
A character pattern used by a computer to perform an individual action including, in some cases, operands or addresses essential to the act.

Machine Language
(see **Machine Code**)

Macro
The keyword that identifies the activity requested of an external system. Part of a macro statement.

Macro Statement

A statement or instruction used as a substitute for, and representing, a set of instructions. In most cases, parameters may be included in order to modify portions of the represented set.

Mark, Question *

Memory
Any portion of a computer or computer peripheral in which information may reside.

Memory, Core *

Minus Sign
The dash character (**-**) that is used as a prefix/infix operator in arithmetic operations.

Modification, Address *

Name
An ordered set of specific characters used to reference a data item or a statement.

Noise
The portion of a signal that contains no information.

Not, Logical *

Not Symbol
The character (**^**) used to represent the logical "not" operator.

No-op
An instruction or statement that can be completely disregarded without affecting the result of a routine in which it may be found. SABRETALK will not produce object code for many types of no-op statements.

Null Statement
A SABRETALK statement that consists only of a statement terminator (**;**), or of a label and a statement terminator.

Number
Used in this guide, loosely, to mean a concept in a number system, a numeral, a numeric, a serial group of digits, etc..

Numeral
A symbol, character or graphic that represents a number.

Numeric
Pertaining to numerals.

Object Code
The representation of a language as it is produced (output) by a Compiler, Assembler, Translator, etc.

Object Program
The output of an Assembler, Compiler or Translator.

Operand
That data item used by an operation process or processed by an operation.

Operation
In SABRETALK, that action performed by a statement operator.

Operator

In SABRETALK, a symbol representing one of a specific group of processes: arithmetic, assignment, relational, concatenation, logical.

Operator, Arithmetic *

Operator, Assignment *

Operator, Comparison *

Operator, Composite *

Operator, Concatenation *

Operator, Logical *

Or, Exclusive *

Or, Inclusive *

Or, Logical *

Or Symbol
The vertical-bar character (**|**) used to represent the logical inclusive-or operator.

Organization, Data *

Output
That which is produced by a program, computer or device and which is, in essence, displayable on paper, film, etc.

Overflow
Usually, the part of the result of an arithmetic operation that will not fit within the limits of the computer's arithmetic registers.

Packed
A statement keyword that allows areas to be positionally adjusted so that they can be more tightly grouped. (See **Aligned**.)

Patch
A temporary insertion into a program. Commonly used for minor corrections when a complete re-compilation or re-assembly is not desired. Sometimes used for the insertion of a unique set of instructions.

Picture Specification
A combination of editing and editing format codes that is used to modify numerals for output printing and/or display.

PL/I
An abbreviation for Programming Language I, a large Compiler adaptable for either scientific or commercial applications and which provides simple or complex levels of programming. SABRETALK is an extended subset of PL/I.

Plus Sign
The symbol (**+**) that is used as an infix operator in addition operations. (Disregarded as a prefix operator.)

Point, Decimal *

Point, Entry *

Point, Fixed *

Pointer
An attribute keyword used to **DECLARE** an area as that which contains an address, usually an address of some larger or major area.

Precision
The accuracy with which a value is represented. (See **Length**)

Prefix
In SABRETALK, the type of operator that precedes one operand, in contrast to the type of operand (infix) that is inserted between two operands. The prefix operators in SABRETALK are the plus (**+**) and the minus (**-**) signs.

PROCEDURE
A labelled unit of sequenced instructions that when given control (with or without parametric information) performs some unique activity. Procedures are a major order of algorithmic internal procedures and of heuristic internal procedures.

Processing, Data *

Program
A complete set of directions and data, in, or convertible into, machine code and which can accomplish a singular task.

Program, Computer *

Program, Control *

Program, Executive
(see **Control Program**)

Program Label
In SABRETALK, the name assigned to a program by attaching it to the first statement.

Program, Object *

Program, Source *

Question Mark
A SABRETALK character that appears only as a character in a character-string.

Radix
The number of digits in a positional numbering system such as the decimal system, binary, hexadecimal, etc. In the decimal system (the radix of which is ten decimal) the number 234 represents:

4
+ 3 times ten
+ 2 times ten times ten

Each successive position has a value ten times greater than the position on its right.

Real-time System
A computer operating system that provides output responses, each within a time period essential to the response.

Record
Generally, a data unit of intermediate size that contains related fields.

Record, Data *

Re-entrant
The property of programs that allows a single copy of the program to concurrently service more than one request by preventing the program from modifying itself during execution.

Reference, External *

Register
One of the group of basic hardware transfer nodes used for arithmetic, logical, relational and assignment operations and for indexing, counting, addressing, etc. All instructions and manipulated data pass through some register. The bit capacity of each register is individually fixed.

Relative Address
A location given as an incremental measurement from a base.

Return, Carriage *

Routine, Control
(see **Control Program**)

Routine, Executive
(see **Control Program**)

Separating Keyword
(see **Separator**)

Separator
In SABRETALK, any character or character group used to keep apart distinct elements of a statement. Examples are the comma, colon, etc.

Separator, Statement
(see **Separator**)

Set
A collection.

Set, Data *

Sign
A representation of positive or negative quantity (**+ -**).

Sign, At *

Sign, Equal *

Sign, Minus *

Sign, Plus *

Significant Digit
Those unit portions of a numeral which represent its true value. Those unit portions of a numeral which have not been changed as a result of manipulation or as a result of insufficient definition.

Source Code
The representation of a language as it is input to a Compiler, Assembler, Translator, etc.

Source Language
(see **Source Code**)

Source Program
A program in source code.

Source Statement
A unit portion of an Assembler or Compiler language.

Space
Referring, in this guide, to print line indexing. Not meant to signify a blank that is a character.

Special Character
The 20 SABRETALK characters that are used as operators.

Specification, Picture *

Statement
A basic problem-oriented language element that is used to delineate data characteristics and problem-solving steps of higher-level languages.

Statement, Assignment *

Statement Body
The portion of a statement that follows an identifying keyword, or that is an assignment.

Statement, Compiler *

Statement, Data *

Statement, DO *

Statement, `IF` *

Statement `LABEL`
The name assigned to an executable statement.

Statement, Null *

Statement Separator
(see **Separator**)

Statement, Source *

Statement Terminator
In SABRETALK, the semicolon (`;`) that signifies the end of a statement.

Step
A common synonym for instruction.

Storage Allocation
The dispensation (and control) of core areas and memory areas. In SABRETALK programs, the responsibility of allocating areas is generally divided between (or shared by) the control program and the Compiler.

String, Constant *

Structure
A logical collection of data items, which may or not have identical data attributes, but which have a hierarchical relationship to one another. A strict tree organization of data where members of the major level are:

named data items, or
a next-lower-level set

and where members of the next (lower) level are:

named data items, or
a next-lower-level set

etc.

Supervising Program
(see **Control Program**)

Suppression
In SABRETALK, the deletion of unnecessary digits in a number, for example, deleting prefix-zeros, or deleting suffix zeros after rounding.

Switch
An address or value used for the selection of alternatives.

Symbol
A graphic sign or set of signs, usually not alphanumeric, that represents some conventional abstract. Examples: **`$ % ->`** .

Symbol, And *

Symbol, Concatenation *

Symbol, Not *

Symbol, Or *

Syntax
The rules of grammar. Computer languages must be necessarily precise and therefore strictly conform to thoroughly defined concepts.

System, Real-time *

Table, Decision *

Terminator
Any construct used to define the end. In SABRETALK, the semicolon (**`;`**) is used as an end-of-statement mark.

Terminator, Statement *

Terms, Factoring-of *

TPF
An acronym for "**T**ransaction **P**rocessing **F**acility", the control or executive program that initiates, terminates and supervises the activities of all other programs.

Transfer Operator
(see **Assignment Operator**)

Truncate
To remove digits from the head and/or tail of a numeral. In SABRETALK, this is often necessary when assigning values to a field of insufficient size.

Upper-case Letters
The capitals of the Roman alphabet.

Value
That which possesses a measurable quality and can therefore be compared.

Value, Absolute *

Variable
In SABRETALK, the contents of a named area to which may be assigned different values.

Word
In SABRETALK, a memory unit of four bytes on a full-word boundary.

Word, Double *

Word, Full *

Word, Half *

Zero Suppression
(see **Suppression**)

INDEX

REVISIONS LOG

These are the revisions added to the current Sabretalk Manual dated 12/18/91. The previous issue was dated 09/05/91.

7.4 CLEAR OPTION ADDED TO COMPILER OPTIONS.

READER'S COMMENTS

These pages are supplied to solicit the reader's comments, suggestions, corrections, questions and requests for missing revisions. Return to:

SYSTEMONE CORPORATION
9300 NW 36th STREET
MIAMI FLORIDA 33178